# 程序员面试攻略

Programming Interviews Exposed

Secrets to Landing Your Next Job

[美] John Mongan
Noah Suojanen
Eric Giguère

赵喜 牛艳萍



人民邮电出版社

本书对程序员面试中的各种注意事项、常见题型和常用解答技巧进行了介绍，书中的面试例题都取材于顶级计算机公司的实际招聘面试题，每道例题的后面都紧跟解决方案的思路和逻辑分析步骤。它不仅能帮助求职者快速复习有关的知识，也对如何给面试考官留下一个良好而又深刻印象的面试技巧进行了指导；而这些能帮助读者获得一份真正的高薪工作。本书适合于所有正在找工作或将要找工作的程序员。

John Mongan, Noah Suojanen: Programming Interviews Exposed: Secrets to Landing Your Next Job (ISBN: 0-471-38356-2).

Authorized translation from the English language edition published by John Wiley & Sons, Inc.



**本书版权登记号：图字：01-2002-5526**

**图书在版编目（CIP）数据**

程序员面试攻略 /（美）摩根（Mongan, J.）等著；杨晓云等译. -北京：机械工业出版社，2003.3

书名原文：Programming Interviews Exposed：Secrets to Landing Your Next Job

ISBN 7-111-08556-6

Ⅰ.程… Ⅱ.①摩… ②杨… Ⅲ.程序设计-自学参考资料 Ⅳ.TP311.1

中国版本图书馆CIP数据核字（2003）第011826号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：武恩玉

北京牛山世兴印刷厂印刷·新华书店北京发行所发行

2003年3月第1版第1次印刷

787mm×1092mm 1/16 ·13.5印张

印数：0 001-4 000册

定价：25.00元

# 前　言

你可能和我们一样不喜欢阅读一本书的前言，但这本书的前言很有用，因此希望大家能把它做为一个例外好好地看一看。如果你对前言确实没有兴趣，我们希望你能看完下面这句话：你对这本书钻研得越深，收获就越大。如果你能把这本书从头读到尾，你肯定会学到不少东西，但是应该在看答案之前先自己动手尝试解决书中给出的练习题。

当你应聘一份程序设计、软件开发或者技术咨询等方面的工作时，招聘方几乎总是会安排一次面试以考察你的程序设计能力。我们写作这本书的目的就是希望能帮助大家顺利地通过这类面试。程序设计面试的目的是为了考察应聘人员的程序设计水平和动手能力，其过程与传统意义上的求职面试并没有多少相似之处，所以传统的应聘秘籍和面试技巧没什么用。程序设计面试题以程序设计题、IQ智力题以及各种与计算机有关的技术性问题为主。本书将对几种常见的程序设计面试题进行分析研究，并通过一些取材真实的面试例题向大家演示一下如何才能最好地回答它们。

看到这儿，读者可能会产生这样几个疑问：做为本书的作者，我们都是些什么样的人？又是谁让我们写这本书的？我们两人都是刚毕业不久的研究生，在过去的几年里，我们参加了许多场面试。从老牌大公司的技术咨询职位到新兴公司的编写设备驱动程序职位，我们面试过的工作岗位可以说是五花八门，这本书就是我们根据自己亲身参加过的各种面试（有成功也有失败）而总结出来的。我们相信，这是写作本书的基础。说实话，我们并不清楚负责招聘工作的人力资源部主管们对程序设计面试工作都有哪些想法（根据我们自己的亲身经历，我们认为现在的程序设计面试工作还有很多地方需要改进。现在的做法过于偏重考察应聘人员解答智力难题或者类似问题的能力，忽视了对其知识面和知识深度的考察，因而很多在软件行业取得成功所必须具备的重要素质都无法得到准确的衡量和评估），我们也不清楚猎头公司将依据什么样的标准来评估应聘人员在程序设计面试中的表现。但在读完本书之后，相信大家都会对美国顶尖的软件和计算机公司里的程序设计面试情况有一个比较实际的了解，并知道自己应该去做些什么以赢得自己想要的那份工作。

需要特别说明的是，本书中的面试题没有一道是我们虚构出来的，这些题全都取材于我们此前参加过的面试。换句话说，类似的题目类型和难度很可能会出现在各位读者今后参加的某次程序设计面试中。但大家同时也要明白：这本书里的例题只是程序设计面试中一些比较有代表性的题目，而不是一个包罗万象的习题集。如

果你想靠死记硬背本书例题和答案的办法来通过程序设计面试，就极可能弄巧成拙。在面试中你可能会遇到这本书里的问题，但你不能抱有这样的幻想。程序设计面试题本来就是千变万化的；而一位聪明的考官在看过本书之后将肯定不会再用本书里的题目。可话又说回来，程序设计面试考题的范围和类型也就是那么多，想变化也变化不到哪里去。只要你不是死抠本书里给出的例题，而是对它们所代表的试题类型进行钻研，那么无论在程序设计面试中遇到什么样的题目，你肯定都能应付自如。

为了帮助大家提高解决问题的能力，我们采用了一种循序渐进的办法。首先，根据实际情况，我们都将在给出面试考题之前先对有关的重要概念加以复习。其次，我们会把解决问题的整个思路向大家解释清楚，而不是简单地直接给出问题的答案。我们发现，从例题的使用方面讲，本书以外的其他教科书或习题集几乎全都采用了另一种套路：先列出一个问题，接着马上给出其答案，然后再解释那个答案为什么是正确的。以我们个人的体会来说，这种套路往往不能给读者以最大帮助：读者能够看懂某个具体的答案并且知道它为什么是正确的，但读者很难了解和把握作者得出有关答案的思路，不容易在遇到与例题类似的问题时做出正确的分析和判断。为了避免上述弊病，本书采用了一种循序渐进的解题方法，而我们希望本书中的解题思路不仅能够让大家知道什么是正确的解决方案，还能让大家明白怎样才能得出正确的解决方案。

只观看而不亲自动手是学不到真本领的。如果你想从本书得到最大的收获，就必须亲自动手去尝试解决书中的每一道例题。我们建议大家采用下面的学习方法：看过例题之后，先把书放在一边并自己动手去寻找答案。如果你在半路上卡了壳，再回过头来研究书中的解决方案。为了让大家开动脑筋，这本书里所有例题的答案全都安排在有关内容的最末尾，所以读者完全不必担心我们会提前“泄密”，让大家“意外地”看到答案。在看过足够多的内容并得到足够多的提示之后，读者应该再次把这本书放在一边并继续开动自己的脑筋。如此重复，直到彻底解决某个问题为止。读者通过自己的努力而得出的解决方案越多，对有关问题的理解也就越透彻。这种学习方法还有另外一个额外的好处，那就是与程序设计面试的实际过程非常相似：你必须完全依靠自己来解决面试考题，但在必要的时候，面试考官会给出必要的提示。

程序设计是一种难度极大的技术性艺术，只通过一本书就把计算机科学和程序设计工作所涉及到的各种细节全都介绍给大家是根本不可能的。因此，阅读本书需要一定基础。我们希望本书读者的计算机知识不低于大学计算机科学系一年级或者二年级学生的水平。具体地说，我们希望读者1）能够熟练地使用C语言进行编程；2）有过使用C++或Java进行面向对象的编程经验；3）了解计算机体系结构和计算机科学理论方面的基础性内容。如果发现自己在上述几个方面有所欠缺，请读者务必在

找工作和参加程序设计面试之前把功课补好。

在本书的读者当中，肯定会有很多人在计算机方面的学识与经验大大超出我们刚才提出的最小要求。如果你就是其中的一员，那你可能对这本书里的一些高级论题——比如数据库、图形处理、并发计算以及Perl语言等——更感兴趣。但千万不要因为自己的经验比较丰富就忽视了基础性的概念和试题。不管你的简历写得多好，面试考官仍会从最基本的问题开始提问。

我们已经尽了最大的努力来保证这本书里的信息是正确无误的。所有的程序代码都经过了编译和测试。但就像读者在你们自己的程序设计工作中遇到的情况一样，程序设计漏洞和错误是在所难免的。一旦发现或者得知这类错误，我们将立刻把它们公布在站点http：www.wiley.com/compbooks/programminginterview/上。

我们相信，本书在你找新工作时一定有所帮助。同时，我们还希望本书中面试例题的分析和解决方案能够对读者找工作时有所启发。如果你想把你对本书的观后感、对书中某个具体例题的想法，或者最近遇到的程序设计面试题等告诉我们，我们将非常欢迎。我们的电子邮件地址是：programminginterview@wiley.com。

预祝大家都能找到一份满意的工作！

John Mongan, Noah Suojanen

---

参加本书翻译工作的人员除封面署名外还有：杨涛、高文雅、王建桥、张玉亭、韩兰、李京山、许玉新、李春卉。

# 目　录

# 求职过程

大多数公司里的求职和招聘过程都差不多。你对可能遇到的情况准备得越充分，你成功的机会也就越大。本章的目的是帮助大家对求职过程有一个比较完整的了解。我们将从因为求职一事而与某家公司开始接触讲起，在看过本书之后，也许你的头几份求职申请表就能让你得偿所愿。在招聘员工的时候，技术型公司的做法与那些传统型公司往往有很大的差异，所以本章内容对那些已经在职场上打拼了多年的人可能会更有帮助。

## 1.1 与公司进行接触

找工作的第一步是与你想加入的公司建立联系。关系网是一个找工作的好途径。告诉你的朋友说你想找一份什么样的工作。哪怕他们没有工作在你想去的那类公司里，他们也可能会知道有哪些公司正在招聘像你这样的员工。如果你的简历是通过诸如“张三的朋友”或者“李四的邻居”之类的关系而传递到负责招聘的那位老兄手里的，那它肯定要比来自陌生人的几百份简历会受到更多的重视和考虑。通过朋友关系认识某公司里的某人之后，下面该怎样做就全看你的了。

直接给那位老兄打电话说“你好，我想和你谈谈找工作的事”的想法是很诱惑人的。对方明白你为什么会打这个电话，所以开门见山未尝不是一个办法。但这个办法有点儿生硬，成功的机会并不会有多大。这种做法往往会让对方觉得你在了解他们的要求之前就已经自认为是他们所需要的人，从而形成一种你这个人有些自以为是或者一厢情愿的印象。因此，为了有一个最佳的结局，最好采取一种比较迂回的策略。你应该先和对方约定一个面谈时间——谁都不愿意在一个不方便的时间去和别人探讨自己的“终生大事”，对吧？即使真的到了面谈的时候，你们之间的谈话也应该先从了解该公司的有关情况开始。如果那个公司听起来的确是个值得一去的好

地方，再向他询问他们公司都想招聘哪些人员。如果你觉得某个空缺正合乎你的心意，就应该向对方说明一下你为什么觉得你本人是一个最佳候选人。最后，别忘了感谢对方抽出时间来和你进行交谈，问问是否可以把你的简历寄给对方，以及要想得到那个空缺是否还需要与其他什么人谈谈。

这种人际交往的手法有着最高的成功率，但其他一些途径也很值得考虑。比如说，当劳动力市场比较紧缺的时候，很多公司都会委托一些人才中介机构——即所谓的“猎头公司”（虽然求职和招聘双方都习惯于使用“猎头”这个词来称呼这类中介机构，但有些从事人才中介活动的人却不喜欢这个称呼。为了保险起见，我们建议你在与人才中介机构打交道的时候最好不使用这个词）——来帮忙物色人才。你可以把自己的情况提供给猎头公司，让他们在得知有适合你的工作空缺时通知你。不过，猎头公司本身也有好有坏，所以你最好多打听打听，看看朋友们有没有什么好的推荐。如果这样还没找到猎头公司的话，你可以到网上去搜索一下猎头公司、职业中介或者就业服务中心等。你可以要求猎头公司向你提供几个人名以便你查询他们的业绩，但因为猎头公司通常会接触大量的人，所以即使业绩很差的猎头公司也往往能向你提供5到10个令人挑不出毛病的客户人名来供你查询，你必须对这一点有心理准备。尽量躲避那些想做你的全权代理或者向你收取费用的猎头公司。有信誉的猎头公司都有这样的自知之明：他们只是你求职过程的一部分，既不应该喧宾夺主，也不应该惟利是图；他们的利润应该来自雇主方面，而不应该来自雇员方面。

在与猎头公司打交道的时候，一定要明白他们的本质。猎头公司应该在他们所推荐的人员被聘用之后才能得到报偿，所以猎头公司的基本想法就是在尽可能短的时间里让尽可能多的人找到尽可能多的工作。要是没有报偿方面的刺激，猎头公司既不会为你去寻找最合适的工作，也不会为企业去寻找最合适的候选人。猎头公司之所以要帮别人找工作，其动机在于赚钱谋生，不是无偿地帮助别人。如果你明白这一点，就不会对他们的所作所为感到惊讶或者失望了。但这并不等于说猎头公司里都是些坏人或者他们的天性就是乘机占申请人或企业的便宜。这么说吧，猎头公司非常有用，只是你不能指望他们会把你的利益摆在他们自己的利益之上而已。

你还可以考虑直接与有关公司进行接触。此时，因特网将是一个最佳的渠道。不管你心里对自己想去工作的公司是否已经有了一些想法，你都应该到网上去找找与你专业对口的那些公司。大多数公司的Web主页上都会有一个供浏览者发送其简历的链接和有关说明。如果你在某公司的Web主页上找到了具体的职务空缺，请仔细阅读有关要求并根据自己的志趣决定是否需要提交一份简历。在许多公司，应聘某个具体职位的简历通常都会被转发到该职位的直接上级经理手里，而那些没有注明具体职位的简历则会被集中到该公司的人力资源数据库里去。如果你在Web主页上找不到提交简历的链接，请再找找上面有没有供你发送简历的电子邮件地址。你

应该把自己的简历以纯文本的形式写在电子邮件里（便于对方直接阅读），同时再发送一份Microsoft Word文档做为附件（便于对方打印一份精美的拷贝）。要知道，并不是每个人都会把他们的Word升级到最新的版本，所以你最好把自己的简历转换为可以在低版本Word里读取的格式。此外，千万要保证你的简历里没有携带宏病毒。不过，像这样直接与有关公司进行联系多少有点无意插柳的感觉，可它既花不了你多少时间也用不着你花费太多的精力，不会给你造成任何损失。

招聘会是一个能让你不需要花费太大的精力就能接触到大量公司的场合。在招聘会上，因为每个公司都会接触到大量的申请人，所以具体到某个公司，你成功的机会并不大。但参加招聘会的公司往往数量众多，所以如果不挑剔的话，你找到一份工作的机会还是比较大的。只要你能在招聘会上多收集一些名片并在招聘会后与有关人等保持进一步的接触，就能让自己从众多的应聘人员中脱颖而出。

你也可以求助于那些比较传统的求职办法，比如报刊杂志上的分类广告和因特网上的招聘数据库等。如果条件允许，你还可以求助于大学里的职业介绍中心、校友会或者行业协会等机构来寻找工作。

## 1.2 筛选面试

如果你的简历打动了招聘方，让他们觉得有必要和你做进一步接触，那么下一步就将是所谓的“筛选面试”了。这种面试通常通过电话进行，持续时间一般是30分钟左右。筛选面试也可能在招聘会现场或者在大学毕业求职会期间的校园里进行。

筛选面试的目的主要有两个。首先，招聘方需要通过这次交谈来确认你愿意从事他们打算聘用你从事的那份工作、你具备从事该项工作所必需的技能、你也能接受该项工作的有关要求和安排，比如工作地点和出差事宜等。如果你们双方能够在这些方面达成共识，面试考官通常会再问你一些专业方面的问题。筛选面试的作用是剔除那些简历里有不实之词或者实际技能无法满足岗位要求的求职者。如果你成功地回答了所有的问题，招聘官通常会在一个星期内再次与你联系，要求你安排时间以参加在该公司办公地点举行的正式面试。

## 1.3 正式面试

你在正式面试过程中的表现是你能否获得一份工作邀约的最关键因素。这种面试通常包含一些技术性的问题，比如编写一段小程序或函数，参加一个计算机编程语言、程序设计技术水平测验，求解一些数学和逻辑智力题，等等。本书的绝大部分内容都是为帮助大家回答好正式面试中的各种问题而准备的。

正式面试通常由3到5个部分组成，每个部分需要30到60分钟，因而整个面试过程通常会持续一个半天或者一整天。你的面试考官通常就是你将要与之共事的团队

成员们——当然前提是你能通过面试并加入这家公司。一般说来，大多数公司都有这样一个原则：即任何一位面试考官都有权否决聘用某位求职者；也就是说，你遇到的每一位面试考官都很重要。有时候，你可能会在同一天内被两组不同的团队面试，而每组面试考官通常会独立地做出关于是否要聘用你的决定。

面试日的午餐通常由招聘方负责。在一家很不错的饭店里吃一顿免费午餐当然是一件好事，但你千万要表现得举止得体，不要因小失大。如果因为就餐时给人留下坏印象而失去了获得一份好工作的机会，可就得不偿失了。要有礼貌，尽量避免酒精饮料和牛排之类的油腻食物。这些守则适用于一切公司聚餐场合，在晚上举行的求职联谊活动也不例外。在晚餐时适当地喝点东西是可以接受的，但一定要有节制——酒量大并不会增加获得工作的机会。

在面试日结束的时候，你通常有机会与该公司的老板进行一次面谈。如果公司老板和你谈话的时间比较长，并且谈话的中心意思是建议你加入他的公司，就说明你在面试中的表现已经打动了他，你极有可能会很快地受到该公司发出的工作邀约。

## 1.4 衣着

传统的面试着装是西服加领带。但如今的技术型公司通常对衣着并没有严格的要求。在某些公司甚至流传着这样的笑话：穿西服的人不是来参加面试的，就是来搞推销的。如果公司上下只有你一个人穿着西服，那它对你可能并没有什么好处。话又说回来，如果你穿的是牛仔裤和T恤，面试考官——哪怕他们自己穿的是牛仔服——又可能会觉得你表现得不够严肃和尊重。一般说来，参加技术型面试的标准衣着是一条免熨的水洗裤、一件带领子的衬衣、再加上一件得体的外套（上面千万不要有金属扣子或金属链条之类的零碎）。除非你参加面试的工作有着商业方面的性质或要求，你一般用不着穿西服打领带。

## 1.5 职业中介

你的面试和工作邀约通常要由一位职业中介人或者一位来自有关公司人力资源部门的代表来安排。如果是这样的话，你参加面试的时间和其他相关事宜——比如你参加面试的旅行和住宿费用等——将由那位中介人来安排和解决。中介人通常不参与最后的聘用决策过程，但他们会把对你的印象和观感传达给那些有决策权的人们。打电话通知你最终结果并安排最终磋商的人通常也是这些中介。

类似于猎头公司的情况，把职业中介人的位置和作用弄清楚对你有着重要的意义。一旦公司决定向你提供一个工作邀约，中介人的工作目标就将转变为怎样让你在最低的薪资水平上接受他们的邀约。职业中介人的收入回报通常与他们实际签定工作合同的人数挂钩的。

职业中介人一般都有着很丰富的经验。他们会大谈特谈某个工作岗位的优势或美好前景，对这个岗位的劣势则往往避而不谈。职业中介人有时会以一种职业咨询师的面目出现，他们会询问你收到的每一个工作邀约，然后向你提供一个精心策划的职业目标分析，让你自己去判断哪一个邀约是最好的。用不着吃惊，这套把戏的结论永远都是该中介人代表的公司所提供的工作邀约是最适合你发展的。

职业中介人通常会拍着胸脯说你关于工作邀约的事情都可以找他去解决。如果你想知道福利或者薪资方面的事情，一般都不会有什么问题，但如果你想了解关于这项工作本身的事情，他们就往往会用一些美好的言辞对你进行误导。中介人对你将要走上的工作岗位往往了解得并不透彻。如果你对工作本身有疑问，中介人是不会有兴趣为你去刨根问底的——特别是那个问题的答案有可能让你拒绝接受那份工作的时候。在遇到这种事情的时候，中介人通常会用一些模棱两可的话或者一些投你所好的答案来敷衍你。如果你想知道某个问题的真正答案，最好的办法是直接去询问你将与之共事的那些人。如果你认为中介人在敷衍你，你完全可以直接找你未来的经理去谈一谈。不过，这种策略是有风险的，你的中介人肯定会对你感到不满，但中介人一般听命于你未来的经理。一般说来，你未来的经理往往要比那些中介人更容易打交道。在职业中介人眼里，你只是求职者之一；但在你未来的经理眼里，你却是一位他挑选出来并准备长期共事的工作伙伴。

有的职业中介人有很强的领地意识，不愿意向求职者透露招聘方的联络办法。为了预防出现这种情况，你应该在参加面试的时候多收集一些面试考官的名片，特别是你未来的经理的名片。这样，你用不着通过你的职业中介人就能直接获得自己想要的信息了。

大多数职业中介人都是有信誉的，你应该尊重他们的劳动并表现得彬彬有礼。但千万不要放松警惕，友好的态度并不等于无保留的帮助，职业中介人的工作目标并不是帮助你，他们的工作目标是让你以最低的薪资水平尽快地与他们所代表的公司签定工作合同。

## 1.6 工作邀约和磋商

如果收到了工作邀约，就说明你已经通过了最困难的阶段，此时你已经拿到了这份工作——如果你愿意的话。可是，游戏还没有最终结束。你找工作的原因是你需要赚钱，而你在这场游戏快要结束时的表现将极大地决定你最终能够得到多少回报。

当你的职业中介人或者招聘经理向你提供一份工作邀约的时候，他一般会同时告诉你该公司打算给你多少报酬，但更常见的情况却是职业中介人或招聘经理在此时会问你想要多少报酬。回答这个问题的办法详见第11章“非技术问题”。

在收到一份详细注明了薪资、签约金和股票期权等细节的工作邀约之后，你需

要认真考虑这份邀约是否合乎你的心意。绝不能在这件事情上将就，也不要当场接受工作邀约。像这种重要的决定至少要花一天的时间来斟酌，一天之内发生的变化往往会大大超出你的意料。

职业中介人往往会采用各种高压手段来诱使你尽快接受工作邀约。他们的常用伎俩之一是告诉你说，如果你想要这份工作，就必须在多少多少天之内接受邀约；另一种伎俩是给你开出一份很高的签约金，而你每拖延一天，这份签约金就会减少一定的数量。千万不要让这些花招干扰了你的决定。如果有家公司真的需要你（它向你发出工作邀约这件事本身已经证明了这一点），那么这些限制和条款就应该是可以磋商的——哪怕职业中介人宣称它们是不能磋商的。如果你的中介人拒绝变通，你就完全可以绕过他而直接与你的招聘经理取得联系。要是这些事情真的无法再变通，你说不定还不想到这家苛刻的公司里去上班呢。

在经过细致的思考之后，如果你认为这份工作邀约已经达到或者超过了你原先的预期，我们就要祝贺你找到一份满意的工作了。如果你对这份工作邀约不是百分之百的满意，你应该尝试去进行一下磋商。很多求职者会想当然地认为自己收到的工作邀约是不能磋商的，他们可能在未做磋商尝试的情况下拒绝了邀约，或者是接受了一份自己并不很满意的工作。其实事实并不像他们想像的那样，几乎每一份邀约都有一定的磋商余地。

永远不要在未做磋商的情况下因为金钱方面的原因而拒绝一份邀约。记住，如果邀约不能让你满意，在拒绝它之前进行一下磋商绝不会损害你的利益。要知道，你手里有最大的王牌——要是事情还没有好转，你就拒绝那份邀约好了；本来就无所得，当然也就无所失。

我们这里要说的是，即使邀约已经达到你的预期目的，也值得再进行一下磋商。只要你在磋商过程中表现出了真诚和信任的态度，并且你的要求也很合理，你就决不会因为试图进行磋商而错过一份邀约。即使出现了最坏的情况，即对方公司拒绝改变它提出的工作邀约条件，你得到的也不会比你开始磋商之前更少。

如果你想就自己报酬与对方进行磋商，请按下面的办法进行。首先，你应该精确地计算出你自己到底想要些什么，如一笔签约金、更高的薪水或者更多的股票期权。把这些东西都计算清楚之后，找个适当的时间给对方公司里有权与你进行磋商的人打一个电话，这个人通常就是签发你手里这份工作邀约的人。不要盲目地乱打电话，这种事绝不能在对方不方便的时候进行磋商。

电话接通以后，你先要感谢对方向你发出了工作邀约，然后再把你为什么觉得不完全满意的地方告诉他。比如说，你可以这样说："非常感谢你给我的工作邀约，但我现在很难决定接受它，因为与我接到的其他邀约比，它没有很大的吸引力。"你还可以这样说："感谢贵公司的工作邀约，但让我接受它却有点困难。因为我通过与同事或者其他公司的谈话了解到这样一个情况，即贵公司的邀约低于目前的行情。"

如果对方让你详细说说哪些公司提出了更高的邀约以及那些邀约的具体数额，或者问到你所说的同事都在哪家公司上班，你可以巧妙地避开这样的问题——你没有义务这样做。你可以这样说："我认为我应该对这些邀约保守秘密，贵公司的邀约当然也包括在内。我想，随便透露这类信息的做法是不恰当的。"

此时，对方或者会问你有些什么想法，或者会告诉你说这份邀约不能磋商。声称邀约不能磋商通常只是一种谈判技巧，而且是一种很难对付的技巧。无论你遇到的是哪种情况，你都应该保持礼貌和尊重的态度并清楚地说出你心目中的邀约条件。对方很少会对邀约条件的变化当场做出拍板，所以你应该感谢对方提出的时间和帮助，然后告诉对方说你会等他在考虑之后再给你答复。

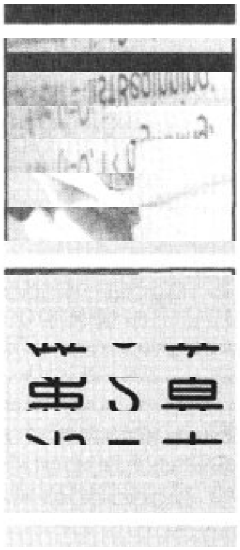
很多人对这种磋商感觉不舒服，特别是在与那些专业化的职业中介人打交道的时候更是如此。要知道，那些中介人每天的工作就是进行这类的磋商。有很多人之所以会接受略嫌不满的邀约，就是为了回避这种磋商。如果你对这种磋商也持同样的态度，那你应该这样鼓励自己：我不会有任何损失，而这种磋商中哪怕最不起眼的胜利也非常值得。如果你用30分钟的电话使你的邀约增加了3 000美元，就相当于每小时赚6 000美元，即使是有名的大律师也挣不到这么多钱呢。

## 1.7 接受或拒绝工作邀约

你与对方公司的磋商迟早会结束，而且你迟早会接受一份工作邀约。在你通知对方公司说你已经接受它们的工作邀约之后，一定要与对方保持联系，把上班日期和签定工作合同等方面的事情确定下来。

对于其他公司发来的工作邀约，你应该以善始善终的态度做出答复。保持友好的联系是很重要的，在人员跳槽频繁的计算机行业更应该如此。在找工作的过程中，你与向你发出工作邀约的公司都进行过联系，这一点是无可否认的。一言不发地把其他公司的联系人抛开是很愚蠢的做法。你应该给其他公司里的联系人发一封电子邮件，把你的决定告诉他们。你还应该给每一位向你发出过工作邀约的招聘经理打个电话，向他们表示感谢并把你的决定告诉他们。比如说，你可以这说："感谢你给我发来工作邀约。我对贵公司的印象非常好，但我觉得它并不是我现在的最佳选择。谢谢你的关照，也感谢你对我的欣赏。"这一方面能体现出你是一位知书达礼的人，另一方面会给对方留下一个好印象，对方可能会这样说："很高兴能认识你，你不能来我们公司真遗憾。如果那家公司的情况有变化，别忘了给我来个电话，也许我们以后还能合作。祝你好运。"

这种结局是最好的。等你下次再想找工作的时候，你就可以从这些地方开始你的新历程。



# 程序设计面试题的解答思路

编写代码是程序设计面试过程中的重头戏，是你展示自己具备胜任有关工作的能力的机会。在解答程序设计面试题时的表现是大多数计算机公司和软件公司决定是否聘用某人的重要标准之一。有很多公司只会给参加程序设计面试人员中不到百分之十的佼佼者发出工作邀约。程序设计面试题通常都有相当的难度。如果每个人（或者绝大多数人）都能迅速回答出某个题目，那么该公司就不会再使用这个题目来做为面试题，因为它根本不能反映出求职者的水平差距。大部分面试题基本上要用一个小时的时间才能得出答案，所以无法立刻找到面试题的解答方法是很正常的，千万不要因此而灰心丧气。要知道，天才毕竟是极少的，其他人并不见得比你强到哪儿去。

提示：程序设计面试题通常都很难！有些题目只是想看你在遇到无法立刻找到其解决方案的问题时会如何应对。

## 2.1 面试过程

在程序设计面试中，你通常要当着面试考官的面来逐个地解答出各道面试题。他会给你一支记号笔和一块白板（或者是钢笔和纸）并让你把有关的代码写出来。面试考官可能会要求你在写出代码之前先对题目本身做出一些分析和说明。最常见的题目是编写一个函数代码，偶尔也会让你编写一个类（class）定义或者一组相关函数。无论如何，编写一些程序代码是避免不了的。

如果你正谋求的工作需要使用某种程序设计语言，那你不仅要对该语言了如指掌，还必须有把握能够用该种语言解决任何问题。如果你正谋求的是一项泛泛的程序设计或者软件开发职位，那你只要精通C语言并熟悉C++就基本上能够过关。面试考官也许会允许你使用Java或Perl等其他主流的程序设计语言。如果你能够选择，请

选择你最熟悉程序设计语言，但对面试考官要求你使用C或C++来解决某些问题的情况也要有所准备。一般说来，如果你使用的是Lisp、Python、Tcl、Prolog、Cobol或者Fortran等非主流程序设计语言，你的面试考官可能会不太满意，但如果你确实是这些语言的专家，问一下面试考官应该不会有什么坏处。在参加面试之前，你应该把自己打算在面试过程中使用的各种程序设计语言都好好地复习一下，准备得越充分，成功的机会就越大。在程序设计语言的选择方面我们还有最后一条忠告（不论对错）：有很多人认为Visual Basic和JavaScript不是一种完备的程序设计语言。如果你应聘的工作不是必须使用这两种语言，那你还是在面试过程中避免使用它们为好。这本书里的解决方案都是用C、C++、Perl或Java编写出来的，其中尤以C语言为重点——它仍是目前各种程序设计面试中最为常用的语言。

你在程序设计面试中写出来的代码很可能是你的面试考官惟一能够看到的代表你编程水平的东西。如果你在面试时写出来的代码质量比较低劣，那你的面试考官很可能会认为你写的代码水平总是这么差。所以，你应该在程序设计面试中努力把你最值得炫耀的代码拿出来。要让自己的写出来的代码既实用又美观，为了做到这一点，多花些时间是值得的。

**提示：** *在参加面试之前一定要对自己可能会用到的程序设计语言进行复习，要把自己最好的代码拿出来。*

程序设计面试题一方面是要考验你编写代码的功力，另一方面是要考验你分析和解决问题的能力。如果面试考官只是想知道你的编程能力，那他完全可以给你一张纸和一个问题后离开考场，然后在一个小时之后再回来看你做得如何。事实上，面试考官想了解的是你将在整个面试过程中表现出来的分析和解决问题的能力。这就使程序设计的答题过程成为你与面试考官之间的一种交流，如果你遇到了困难，面试考官通常会通过一些暗示或者提醒来帮你找到正确的答案。当然，你在解答面试题时需要的外来帮助越少，你表现的也就越出色；可即便如此，展现出你睿智的思维过程并对面试考官的提示做出聪明的回应仍将是十分重要的。如果你还额外知道一些与面试题有关的信息——哪怕这些信息与摆在你面前的面试题没有很直接的联系，你也应该把握适当的机会去高谈阔论一番以进一步表明你的计算机水平。这是你表现自己不是一个只知埋头编写代码的死脑筋的好机会。你应该把你严密的逻辑思维能力、丰富的计算机知识以及流畅的人际沟通技巧展示给你的面试考官。

**提示：** *要多说话！要随时向面试考官解释你正在做些什么。*

面试题通常是越往后难度越大。这虽然不能算做是一条定律，但当你正确解答了的题目越多的时候，面试考官后面给出的题目就肯定会越难。一般说来，面试考官会彼此交流他们给你出了哪些考题，以及其中有哪些是你解答出来了的、还有哪

些是你尚未解答出来的。如果你在早些时候的面试里把所有的面试题都解答了出来而在稍后遇到了一些很难解答的题目，就说明早些时候的面试考官对你的答题能力有了比较深刻的印象并通报给了其他面试考官。

## 2.2 关于面试题

程序设计面试题本身具有比较特殊的要求。首先，它们必须简短到足以让优秀的求职者迅速做出解释和解答；其次，它们又必须复杂到不能让所有的求职者都毫无困难地解答出来。因此，你不太可能会遇到现实世界中的问题。现实世界中的问题几乎都需要至少三个小时的时间去解释，需要至少一天的时间去分析现有代码，再需要至少一个星期的时间去得到解决。总之，现实世界中的问题不适合用做考察求职者水平的程序设计面试题。反过来说，程序设计面试题通常都需要你动点脑筋并使用一些不常用的编程技巧。

程序设计面试题通常会禁止你使用那些最容易想到的办法或者禁止你使用最理想的数据结构。比如说，你可能会遇到这样一道面试题："编写一个函数来比较两个整数的大小，但不得使用任何比较操作符。"（提示：如果你不知道该怎样解答这个问题，不妨试试二进制位操作符。）说老实话，这是一个非常愚蠢和无聊的题目。所有程序设计语言几乎都会提供比较两个整数大小的直接办法。但是，如果你用下面这番话来做为回答可就麻烦了："这是一个愚蠢的问题；我怎么也得用到等号吧。这个问题我可没见过。"不客气地讲，要是你真的这么回答的话，你将立刻被淘汰出局。在遇到这类情况时，你可以考虑采取这样的对策：先说明这道面试题其实还有更好的解决方案，只是因为题目要求的缘故，你不能那样做；然后，再按面试题目的要求另辟出路。比如说，如果面试考官要求你必须使用哈希表（hashtable）来解答某个题目，你可以这样说："这个问题用二元搜索树（binary search tree）更容易解决，因为用二元搜索树来找最大值要容易得多。不过，用哈希表也不是不可以。我想这样来解决这个问题……"

> 提示：程序设计面试题要么有着古怪的限制性要求，要么需要使用不常用的语言功能；它们通常都显得很愚蠢和无聊。不过，这也正是你必须遵守的游戏规则。一定要随机应变。

## 2.3 答题方法

如果根本看不懂题目，你也就根本无法解答。面试题里往往隐含着一些假设，而面试考官的解释要么非常简短，要么很难理解。要是你连面试题说的是什么都弄不明白的话，展示自己编程水平的事也就无从谈起。请记住我们的忠告：如果你对

面试题本身有什么不清楚的地方，千万不要犹豫，你应该赶快向面试考官问个明白。如果你还没有把题目弄明白，千万不要急于答题。

在把题目弄明白之后，你应该先设计一个例子。这个例子往往能帮你找出问题的关键，或者帮你搞清楚你尚未透彻理解的地方。从例子开始入手往往还能展示出你的逻辑思维清晰。当你无法立刻看出题目解答方法的时候，先举出一个例子将非常有用。

**提示：** 在把题目弄清楚之前千万不要开始答题。从一个例子开始入手能进一步加深你对题目的理解。

在举出一个例子之后，你应该把注意力迅速转移到解决这个问题的算法上来。这往往要多花费一些时间，说不定还需要你再多找出几个例子来。这正是面试考官希望看到的情况。如果你只是静静地看着白板发呆，面试考官就无法知道你是在思考，还是大脑一片空白。因此，你应该不停地与你的面试考官进行交谈，告诉他你正在做什么事情。比如说，你可以这样说："我正考虑是否应该先把这些值保存到一个数组里，然后再对它们进行排序。这个办法好像行不通，因为数组里的元素是无法用它们的取值来查找的……"这番话展示了你的本领，而这正是程序设计面试的目的所在，而且还极有可能"诱使"面试考官对你做出一些提示。面试考官可能会回应你说："你已经很接近正确的解决方案了。你真的需要用数组元素的取值来查找它们吗？也许你……"

解答问题可能要花费你很长的时间。如果你耐不住性子，就极有可能在完整地构思出题目的解答方法之前就急于动手编写代码。一定要抵制住这种诱惑。你不妨自己去想一想：一个人是考虑了半天才开始动手，但一出手就写出了正确的代码；另一个人则是立刻就开始编写代码，但他的代码是边写边出错，而且你还看不出他有没有头绪；要是让你来选择，你愿意与哪个人共事？这个问题并不难回答，对吧？

在确定下算法并考虑好如何用代码来实现这个算法之后，你应该向面试考官解释一下你的解决方案。这将使他有机会在你开始编写代码之前对你的解决方案做出评估。你的面试考官可能会这样说："听起来很不错，再让我们看看你写出来的代码吧。"他也可能会这样说："这好像有点不妥，因为哈希表里的元素是不可能用这种办法找出来的……"不管面试考官怎么说，你都能获得一些宝贵的提示。

在编写代码的过程中，随时对自己写出来的代码做出解释是很重要的。比如说，你可以这样讲："现在，我把数组全部初始化为零……"这种做法能够让面试考官跟上你的思路，从而更好地理解你写出来的代码。

**提示：** 在动手编写代码之前和编写代码的过程中，应该随时向面试考官解释你正在做的事情。要多说话！

在答题过程中，不要害怕向面试考官求助。一般说来，如果你的查问是那些需要从某些参考大全里查出答案的问题的话，你就不会受到“惩罚”。当然了，你肯定不能提出诸如“这个题目应该怎样解答”之类的问题。但像下面这样的提问应该是可以接受的：“我有点记不清了——能不能告诉我用来输出八进制数字的printf格式字符串是哪个？”用不着问这种问题就知道八进制数字的输出办法当然是最好的了；即便是问了这类问题，也总比你写错了代码要好；所以你千万不要有所顾虑。

代码编写完毕之后，应该立刻找一个例子来验证这段代码的正确性。这个步骤一方面有助于清晰地证明你的代码至少适用于一个特例；另一方面则表明你有着严密的逻辑思维，习惯于检查自己的代码成果并进行纠错。那个例子还有助于你把自己解决方案里的小漏洞找出来。

最后，你应该确保你写出来的代码已经把各种出错情况和特例都考虑周全了。有不少出错情况和特例是很容易在编写代码时遗漏的，在面试时遗漏了这些特例意味着你在实际工作中也可能会遗漏它们。举例来说，如果你分配了动态内存，就必须检查这个内存分配操作是否成功。此外，一定要检查自己的代码里有没有引用空（NULL）指针，有没有对空数据结构进行操作。正确地处理好这些“小”事是非常重要的，它一方面会给面试考官留下深刻的印象，另一方面能保证你干净彻底地解决了这道考题。

**提示：** 找个例子来验证自己写出来的代码，并检查自己的代码是否把各种出错情况和特例都考虑周全了。

在你用例子验证完自己的代码并完成这段代码的正确性检查工作之后，你的面试考官往往会就你的代码提出一些问题。这些问题通常集中在这段代码的运行时间、替代方案及其复杂性等几个方面。如果面试考官没有向你提出这些问题，那你应该主动地谈论一些这方面的话题以表明你考虑到了这些事情。举例来说，你可以这样讲：“这种实现方法的运行时间是线性的，它是这一问题的最佳解决方案，因为我必须检查所有的输入值。动态内存分配操作多少会降低一点它的运行速度；递归算法也增加了一些开销……”

## 2.4 遇到疑难时

一般说来，你肯定会在解答某个题目时遇到麻烦。这是面试考官“蓄谋已久”的事情，也是面试过程的一个重要组成部分。面试考官就是想看到你在遇到无法立刻找出其答案的问题时会有什么样的反应。最糟糕的反应有两种，一种是颓然放弃，另一种是望着天空发呆。在遇到这种情况的时候，你应该表现出对这个问题的兴趣和锲而不舍地解决这个问题的决心。如果没有别的办法可想，你应该回到先找一个

例子的老路上来：一边用这个例子往前推进，一边分析自己正在进行的操作，再把思路从这个具体的例子延伸到普遍的情况上去。你也许需要使用非常具体的例子，这是谁都能理解的。

**提示：** 如果没有别的办法可想，你应该回到先找一个例子的老路上来：一边用这个例子往前推进，一边分析自己正在进行的操作，再把思路从这个具体的例子延伸到最终的解决方案上去。

另一种思路是试着换用一种不同的数据结构。换用链表（linked list）、数组、哈希表或者二元搜索树说不定就能让问题迎刃而解。如果面试考官要求你必须使用某种不同寻常的数据结构，那你应该设法把它与你比较熟悉的数据结构之间的异同点找出来。恰倒好处的数据结构往往能大大简化问题的难度。

在遇到比较棘手的题目时，你还应该多考虑考虑某种程序设计语言不太常用或者比较高级的特色功能，比如二进制位操作符、复合型数据类型（union type）、复杂的指针映射、不常用的保留字等。在很多时候，解答高难度题目的关键就需要这几类不常用的功能。

**提示：** 解决高难度题目的关键往往是另外一种不同的数据结构或者程序设计语言中某些不常用的功能。

有时候，甚至在你还没觉得卡了壳时，问题却已经出现了。例如，你在完成某项具体操作时忽略了某个精巧或者明显的办法，结果是使用了太多的代码。程序设计面试有一个几乎是放之四海而皆准的规律，即编程考题的最佳答案通常都非常短小精练，很少需要你写出15行以上的代码，并且几乎从不超出30行以上。要是你写的代码在行数上超过了这两个数字，你就应该提醒自己是否已经走错了方向。

说到编写代码，我们还有最后一句忠告：你经常需要把一个值与“NULL”或“0”进行比较。请看下面的例子（这个例子至少适用于C或C++），你可以这样写：

```
if (elem != NULL) {
```

也可以这样写：

```
if (elem) {
```

这两条语句在编译器看来是完全等价的，但程序员们却对其优劣有各自的看法。有些程序员认为第一种写法更明确，程序代码应该写成这种易于阅读和理解的形式；可另一些程序员却认为因为这种情况非常多见，所以第二种写法也完全可以接受。从面试考官的立场上看，这两种写法在技术上都是正确的。但细论起来的话，第一种写法说不定会让面试考官疑虑你是否了解与“NULL”进行比较的必要性，第二种写法则打消了面试考官的种种疑虑并会把你看做是一位编程老手。因此，在面

试时还是采用第二种写法比较好，如果你还能顺便说明一下你的做法完全等价于与“NULL”或“0”进行比较，效果就更完美了。

## 2.5 对解决方案进行分析

面试考官经常会就你代码实现的执行效率提出一些问题，其中最为常见的情况是要求你把你的代码实现与其他解决方案进行对比并说出它们各自的优缺点和适用场合。面试考官的问题通常都集中在动态内存和递归算法的使用方面。

衡量执行效率的重要标准之一是对代码执行时间进行的估算，而程序员们最熟悉的代码执行时间估算方法则非“大O记号法”（big-O analysis）莫属。这种记号法向我们提供了一个能够用来对比两种算法各自的执行时间的计量尺度。“大O记号法”的正式定义涉及到了很高深的数学知识，而本书在这方面的的讨论则主要以实用和易于理解为出发点。如果你对“大O记号法”很熟悉，那你不妨把下面的内容当做是一次简单的复习；如果你对“大O记号法”不熟悉，那就要请你通过下面的内容快速学习一下代码执行时间分析的要点了。

“大O记号法”针对的是算法的执行时间，它把算法看作是一个函数，而这个函数又有着一定数量的输入。让我们用一个简单的例子来开始我们对“大O记号法”的讨论吧。给定一个元素为数值的数组，要求你写一个函数把该数组中的最大值找出来；这个数组的元素个数是$n$。我们至少可以用两种方案来实现这个函数，而这两种方案实现起来都不难。我们先看第一种方案，你可以在遍历数组元素的同时把当前最大值随时找出来并保存在一个变量里，当你完成遍历时，保存在该变量里的元素肯定就这个数组的最大值。我们给这第一种实现方案起个名字叫“CompareToMax”，它的代码应该是下面这样的：

```
/* Returns the largest integer in the array */
int CompareToMax(unsigned int array[], int n)
{
    unsigned int curMax, i;

    /* Make sure that there is at least one element in the array. */
    if (n <= 0)
        return -1;

    /* Set the largest number so far to the first array value. */
    curMax = array[0];

    /* Compare every number with the largest number so far. */
    for (i = 1; i < n; i++) {
        if (array[i] > curMax) {
            curMax = array[i];
        }
```

```
    }
    return curMax;
}
```

再来看第二种方案，这个函数将把数组中的每一个值与其他所有值进行比较；如果其他值全都小于或者等于某个值，后者肯定就是我们要找的那个最大值。我们给这第二种实现方案起个名字叫“CompareToAll”，它的代码应该是下面这样的：

```
/* Returns the largest integer in the array */
int CompareToAll(unsigned int array[], int n)
{
    int i, j, isMax;
    /* Make sure that there is at least one element in the array. */
    if (n <= 0)
        return -1;

    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            isMax = 1;
            /* See if any value is greater. */
            if (array[j] > array[i])
                isMax = 0; /* array[i] is not the largest value. */
        }
        /* If isMax == 1, no larger value exists; array[i] is max. */
        if (isMax)
            return array[i];
    }
}
```

上面这两个函数都能正确地找出数组中的最大值，可哪个函数的执行效率更高呢？如果你想一丝不差地让函数CompareToMax和CompareToAll的执行速度分出高下，那么最精确的办法莫过于对它们进行速度测试。不过，在开发程序的时候，把每一种可能会用到的算法的执行速度都测试一遍的想法是不切实际的。你需要一种能够在具体实现某一算法之前估算出其执行性能的手段，“大O记号法”正是你想要的东西：它能把不同算法的相对性能估算出来并加以比较。

“大O记号法”把输入数据的规模设定为“$n$”。就上面给出的数组例子而言，这个“$n$”代表着数组中的元素个数。在其他场合里，这个“$n$”所代表的可能是某个链表中的结点个数、可能是某数据类型里的位个数、还可能是某哈希表里的数据项个数。在把“$n$”所代表的输入项的含义确定下来之后，下一步就是估算这“$n$”个输入项到底被某个特定的算法使用了多少遍。上一句话里的“使用”是一个含义不确定的词，它在不同的算法里往往会有不同的解释。比较常见的“使用”情况有：给输入值加上一个常数、创建一个新的输入项、或者删除一个输入值、等等；这些

具体的操作动作在“大O记号法”看来是没有分别的。就函数CompareToMax和CompareToAll而言，“使用”的意思就是把数组中的一个值于另一个值加以比较。

在函数CompareToMax里，每个数组元素都只与一个最大值进行了一次比较；也就是说，$n$个输入项中的每一个只被使用了一次，它们总共被使用了$n$次。“大O记号法”把这种情况表示为“O($n$)”。O($n$)通常被称为线性时间。大家可能已经注意到了，除每个元素被使用了一次之外，还有一个用来确保数组不为空的条件语句和一个用来对curMax变量进行初始化的附值语句。因此，把CompareToMax称做是一个O($n$+2)函数似乎要更精确一点儿。不过，因为1）“大O记号法”给出的是接近于极限情况的执行时间，即执行时间在“$n$”值非常巨大时的极限值；2）当“$n$”趋近于无穷大时，“$n$”和“$n$+2”之间的差异完全可以忽略不计；所以“大O记号法”将略去所有的常数项。类似地，对一个执行时间为“$n+n^2$”的算法来说，当“$n$”足够大时，“$n^2$”与“$n+n^2$”之间的差异也可以忽略不计。换句话说，在使用“大O记号法”的时候，最高次幂以外的其他项全都可以忽略不计，只保留当“$n$”变得很大时的最大项即可。就上面给出的数组例子而言，最高次幂就是“$n$”，所以CompareToMax函数的执行时间将被表示O($n$)。

CompareToAll函数的分析过程要稍微复杂一些。首先，你得先对那个最大值在数组中的位置做出一个假设。我们现在先假设这个最大值位于数组的末尾。在这种情况下，这个函数将把$n$个元素中的每一个与另外$n$个元素逐一进行比较，即总共要使用“$n \cdot n$”次，所以它是一个O($n^2$)算法。

截止到目前的分析表明，CompareToMax是一个O($n$)算法，CompareToAll则是一个O($n^2$)算法。不难看出，当数组元素的个数逐渐增大时，CompareToAll函数中的比较操作次数将大大超过CompareToMax函数中的情况。如果某个数组里有30 000个元素，那么CompareToMax函数将进行30 000次比较操作，而CompareToAll函数则要进行900 000 000次比较操作。CompareToMax函数显然要快得多，因为它需要执行的比较操作次数要少30 000倍。事实上，在某次速度测试中，CompareToMax花费了不到0.01秒的时间，而CompareToAll花费了23.99秒。

有人认为这种对比有故意“欺负”CompareToAll的嫌疑，因为我们把最大值安排在了数组的末尾。这一点我们不想否认，它正好引出了最佳执行时间、平均执行时间、最差执行时间等重要概念。CompareToAll的分析结论是基于一种最差情况——即最大值位于数组末尾的情况——而得出的。我们现在对平均情况——即最大值出现在中间位置的情况——做一下分析。通过计算，我们知道它将进行“$n(n/2)=n^2/2$”次比较操作，所以它的执行时间将是O($n^2/2$)。这个“$\frac{1}{2}$”因子又有什么作用呢？每一个比操作所花费的时间主要取决于从函数代码转换出来的机器指令以及CPU执行这些机器指令的速度，所以这个“$\frac{1}{2}$”并不会使事情发生根本的改变。某些O($n^2$)级

的算法说不定比另一个O($n^2$/2)级的算法执行的还要快。在“大O记号法”里，你应该舍弃所有的常数因子，所以函数CompareToAll的平均情况并不比它的最坏情况好多少，它仍是一个O($n^2$)算法。

CompareToAll在最佳情况下的表现要优于O($n^2$)。此时，最大值将出现在数组的开始。这个最大值将与数组中所有的其他值只比较一次，所以它的执行时间将是O($n$)。

再来看CompareToMax函数，它的最佳执行时间、平均执行时间、最差执行时间都是一样的。不管数组中的元素怎样排列，这个算法永远是O($n$)级的。

用“大O记号法”来分析执行时间的一般步骤如下所示：

1) 确定输入数据集并确定“$n$”所代表的含义。

2) 把算法执行的操作次数用这个“$n$”表示出来。

3) 去掉其他项，只保留最高次幂。

4) 去掉所有的常数因子。

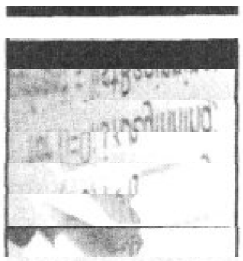
有一个情况必须引起大家的注意。你可以像下面这样对CompareToAll函数进行优化：你不需要把每一个数值与其余的每一个数值进行比较，你只要把它与数组中排列在它后面的那些元素进行比较就可以了。这样做的依据是：先于当前数值的每一个数组元素都已经与当前元素进行过比较了。也就是说，即使你只把某个数组元素与排列在它后面的那些数组元素进行比较，这个算法仍将是正确的。那么，改进后的代码实现在最差情况下的执行时间又是多少呢？第一个数组元素需要与另外$n$个数组元素进行比较，第二个数组元素需要与$n-1$个数组元素进行比较，第三个数组元素将进行$n-2$次比较，最终的总和将是“$n+(n-1)+(n-2)+(n-3)+\cdots+1$”次。这是一个很普通的数列，它的前$n$个数字之和的计算公式是：

$$(n^2/2)+(n/2)$$

因为上面这个公式里的最高次幂是$n^2$，所以它的执行时间仍将是O($n^2$)。

执行时间最快的算法是O(1)级的。这个记号表明有关算法的执行时间是一个常数。也就是说，不管输入数据的规模有多大，有关函数的执行时间都将是完全一样的；也许那个函数根本就没有输入数据。

本书中的大多数编程试题都附带有相应的执行时间分析。希望这些例题能够加深你对这一问题的理解。



# 链　表

我们为什么要花一整章的篇幅来讨论链表（linked list）这种动态数据结构范畴内最没有用处东西呢？是这样的，我们之所以要对链表进行深入的讨论，是因为程序设计面试考官最喜欢用这种动态数据结构来“刁难”那些求职者。别忘了，程序设计面试往往会持续一个小时左右，在这么长的时间里，面试考官至少会向求职者提出两到三个问题。做为求职者，你应该有需要拿出20到30分钟的时间来回答面试考官提出的问题的心理准备。链表是一种相对比较简单的数据结构，你只需稍加练习，就能在10分钟左右的时间内在半张纸上用链表实写出一段比较复杂的代码来，从而留出很多的时间供面试考官们发问。反观哈希表等比较复杂的数据结构，它们往往要花费更长的时间才能实现出来，留给面试考官向你发问的时间当然就要少的多了。此外，链表在实现过程中的变化余地比较小，面试考官只要简单地说一声“链表”就足够了，用不着再多浪费时间去讨论和澄清具体的实现细节。从另一方面讲，链表的复杂程度并不亚于其他动态数据结构，面试考官完全可以用它来构造出一些极富挑战性的试题来。

与其他的动态数据结构相比，现实世界里的程序开发很少会用到链表，所以你很可能对链表这东西运用得并不熟练。因为面试考官很难判断出你是刀法生疏还是根本就不会耍大刀，所以我们首先复习一下有关链表的知识。如果你对链表已经了如指掌，那就请你直接跳到例题部分去做练习好了。

## 3.1　单向链表

面试考官所说的“链表”通常都指的是单向链表（singly linked list）。单向链表（如图3-1所示）由一组数据元素构成，每个数据元素都带有一个指向下一个数据元素的引用链接指针（即后指针）。最后一个元素的后指针将被标记为表明该元素是链表

最后一个元素的样子。在C语言里，链表最后一个元素的后指针将被设置为空指针“NULL”。单向链表各元素的后指针和数据都是绑定在一起的，通常用结构（如C语言里的“struct”）或类（class，如C++或Java）来实现。下面是用C语言对一个整数链表做出的类型声明：

```
typedef struct elementT {
    int data;
    struct elementT *next;
} element;
```

与链表有关的程序设计面试题通常都要求用C语言来解答，偶尔会要求使用C++。之所以会选择C语言，是因为其他语言大都提供有功能更为强大的动态数据结构做为其基本数据类型（比如Perl）或者标准函数库（比如Java）。我们相信，如果你在使用C语言编写程序时都不经常用到链表，那你在使用内建有更强功能的其他语言编写程序时就更不会去考虑使用链表了。

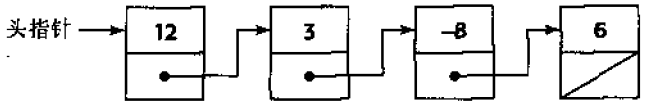



图3-1 单向链表

不管你使用的是哪一种程序设计语言，在实现单向链表的时候都必须留意它的各种特例和容易忽略的编程陷阱。因为单向链表中的引用链接只包括指向下一个元素的指针，所以你只能沿着一个方向对链表进行遍历，要想完整地遍历一个单向链表，就必须从它的第一个元素开始。换句话说，如果你想对单向链表中的所有元素进行某种操作，就必须有一个指向该链表第一个元素的指针。因此，人们常常用“链表”这个词来称呼那个指向某链表第一个元素的指针。比如说，当人们说某个函数需要一个链表来做为输入参数的时候，他们的实际意思其实是说这个函数需要用一个指向某链表第一个元素的指针来做为输入参数。

### 3.1.1 头指针的修改

这个概念引出了链表实现过程中第一个重要的注意事项。头指针是指向链表第一个元素（即头元素）的指针。因为遍历整个链表的操作必须通过头指针来完成，所以无论是添加一个新的头元素还是删除那个旧的头元素，你都必须对头指针做出相应的修改。修改头指针这个操作本身并不难，但当你需要在一个子函数的内部对头指针做出修改时（这是相当常见的情况），问题就出现了——需要你修改的并不是链表的头指针在这个子函数里的局部副本，而是位于其父函数内的那个真正的头元

素指针。我们先来看一段考虑不周的代码，它没能正确地改变父函数里的头指针：

```
int BadInsert(element *head)
{
    element *newElem;
    newElem = (element *) malloc(sizeof(element));
    if (!newElem)
        return 0;

    newElem->next = head;

    /* Incorrectly updates local copy of head.
     * Calling code retains the old value for the first element
     * pointer, so it now points at the second element of the list.
     */
    head = newElem;
    return 1;
}
```

用C语言来修改头指针的正确做法是把一个指针传递给头指针；只有这样，你才能把父函数里的头指针修改为指向新的头元素，如下所示：

```
int Insert(element **head)
{
    element *newElem;
    newElem = (element *) malloc(sizeof(element));
    if (!newElem)
        return 0;

    newElem->next = *head;

    /* *head gives the calling function's head pointer, so
     * the change is not lost when this function returns
     */
    *head = newElem;
    return 1;
}
```

**提示：** 有可能会改变链表第一个元素的任何子函数都必须把一个指针传递给头指针。

### 3.1.2 遍历

对链表进行的操作并不仅仅局限于头元素，你迟早需要对链表中的其他元素进行处理。如果你想对链表中不是第一个元素的其他元素进行操作，必须先通过链表的遍历功能到达那个元素才能对它进行处理。如果你在遍历链表的时候忘了检查是

否已经到达链表的末尾，就可能会遭遇到引用一个空（NULL）指针的危险。请看下面这个示例函数，你想通过它来到达链表的第六个元素：

```
element *FindSix(element *elem)
{
    while (elem->data != 6) {
        elem = elem->next;
    }
    /* Found elem->data == 6 */
    return elem;
}
```

这个函数可以工作得很好，但前提是这个链表至少要有6个元素才行。如果不是这样，那么当你的遍历操作越过最后一个元素时，elem就会设置为“NULL”，从而导致while循环的循环条件对空指针“NULL”进行比较，使整个程序将陷入崩溃。为了避免这个陷阱，你可以用while循环的循环条件来查验遍历指针，确保遍历操作在到达链表末尾的时候会戛然而止；如下所示：

```
element *FindSix(element *elem)
{
    while (elem) {
        if (elem->data == 6) {
            /* Found elem->data == 6 */
            return elem;
        }
        elem = elem->next;
    }
    /* No elem->data == 6 exists */
    return NULL;
}
```

在编写与链表的遍历操作有关的函数时，千万不要忘记我们在此给出的通用思路。

**提示：** 在遍历链表的时候，千万不要忘记检查自己是否已经到达了它的末尾。

### 3.1.3 插入与删除

单向链表中的元素只能通过各元素指向下一个元素的next指针组成这个链表，所以要想在链表半中间位置处插入或者删除元素，就必须对该位置前一个元素的next指针进行修改。也就是说，往链表里插入一个元素必须知道两个指针，一个指向位于插入点之前的元素，另一个指向位于插入点之后的元素；从链表里删除一个元素也需要知道两个指针，一个指向将要被删除的元素，另一个则指向位于删除点之后的元素。不过，因为链表中前一个元素的next指针能够提供指向后一元素的指针，所以

只要知道插入点或删除点前一个元素的指针就足以完成操作。如果面试考官只给出了指向后一个元素（即删除操作中的删除点或插入操作中位于插入点后面的那个元素）的指针，事情就比较麻烦了；没有什么好办法能让我们迅速地把前一个元素确定下来，你只能对链表进行遍历了。下面这段示例代码可以用来确定删除点前面的那个元素。

```
int DeleteElement(element **head, element *deleteMe)
{
    element *elem = *head;

    if (deleteMe == *head) { /* special case for head */
        *head = elem->next;
        free(deleteMe);
        return 1;
    }

    while (elem) {
        if (elem->next == deleteMe) {
            /* elem is element preceding deleteMe */
            elem->next = deleteMe->next;
            free(deleteMe);
            return 1;
        }
        elem = elem->next;
    }
    /* deleteMe not found */
    return 0;
}
```

**提示：** 如果要在单向链表中删除或者插入一个元素，就必须知道指向删除点或插入点前面那个元素的指针才行。

链表元素的删除操作本身还存在着下面这样的问题。假设你想释放链表中的全部元素，比较容易想到的办法是用一个指针来遍历整个链表并在遍历的同时释放途经的各个元素。但这种办法是有疑问的：你是先前进到下一个元素，还是先释放当前元素呢？如果是先前进到下一个元素，你将无法释放当前元素，因为你已经越过了当前元素而到达了它的后一个元素；如果是先释放当前元素，你将无法前进到下一个元素，因为当前元素被释放之后，你也就无法读取它的next指针了。因此，最完善的解决方案是像下面这个示例那样使用两个指针：

```
void DeleteList(element *head)
{
    element *next, *deleteMe;
```

```
    deleteMe = head;
    while (deleteMe) {
        next = deleteMe->next;
        free(deleteMe);
        deleteMe = next;
    }
}
```

**提示：** 单向链表的元素删除操作至少要使用两个指针才能正确完成。事实上，单向链表的元素插入操作也需要两个指针才能正确完成，它们一个负责给出插入点在链表中的位置，另一个则指向由内存分配调用返回的新元素；只是人们在进行插入操作时很少会像在进行删除操作时那样忘记需要使用两个指针而已。

## 3.2 双向链表

双向链表（doubly linked list）消除了单向链表大部分的不足之处（如图3-2所示）。双向链表与单向链表的区别在于前者的每一个元素都有两个引用链接指针，它们一个指向前一个元素（即前指针，链表第一个元素的前指针通常被设置为空指针“NULL”），另一个则指向后一个元素。新增加的指针使我们能够沿两个方向对链表进行遍历，并且以任意元素为出发点都能经过链表中的每一个元素。分别指向前、后元素的指针使我们很容易在双向链表中找出位于任一元素之前或之后的元素，链表元素的插入和删除操作都变得轻而易举了。具体地说，你只需使用一个指针变量就能完成链表元素的删除操作，因为你可以用将被删除的那个元素的前指针来确定位于它前面的那个元素。与双向链表有关的问题很少会成为程序设计面试中的考题。用单向链表难以解决的问题对双向链表来说往往是小菜一碟。从另一方面讲，如果某个问题用双向链表也很难解决的话，采用增加链表复杂程度的办法去对付它也就没有多大的必要了——选用其他类型的数据结构往往会更有效。

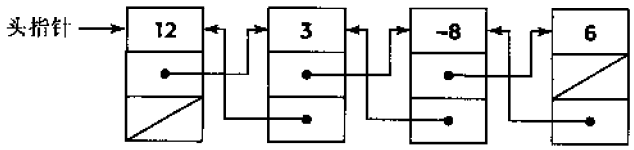



图3-2　双向链表

## 3.3 循环链表

链表家族的最后一位成员是循环链表（circular linked list），它是单向链表和双

向链表的变体。循环链表的基本特点是有头无尾，所以循环链表的遍历操作的主要难点是如何检测出它的末尾以避免遍历操作陷入死循环。虽说循环链表具备一些非常有趣的特点，但它们很少会出现在程序设计面试的试题当中。

链接型元素更为复杂的拓扑结构包括树（tree）、图（graph）、和网络（network），我们将在第4章“树和图”中对它们进行讨论。

## 3.4 面试例题：堆栈的实现

- **请对堆栈这种数据结构做出评论。用C语言来实现一个堆栈，你可以选用链表或动态数组来实现你的堆栈；并请对你的决定做出解释。你为堆栈设计的程序接口必须完备、规范、和易于使用。**

这道面试例题能够对求职者进行三方面考察：1）对基本的抽象数据结构（堆栈）和底层数据结构（链表和动态数组）的理解和掌握程度；2）用编程语言实现和处理这些数据结构的能力；3）为一组相关例程设计一套规范化程序接口的能力。

堆栈是一种“后进先出”（last-in-first-out，LIFO）的数据结构。也就是说，当你从堆栈上取出一个元素的时候，你得到的将是那个最后进入堆栈的元素。堆栈这种数据结构特别适用于那些由多层子任务构成的任务。堆栈比较常见用法有：1）保存子例程的返回地址、参数、局部变量；2）用编译器分析语法时保存各种语法元素记号。往堆栈里添加元素和从堆栈里取出元素的操作通常被形象地分别称为“push”（压入）和“pop”（弹出）。

我们已经在本章前半部分内容里对链表进行了介绍。动态数组是一种总长度会随其中元素个数的增减而做出相应改变的数组，对动态数组的详细讨论请参阅第5章“数组与字符串”中的有关内容。动态数组与链表的主要区别是前者的随机存取性（你可以立即存取任一元素），但因为堆栈上的操作永远出现在这种数据结构的尾部（即堆栈的栈顶），所以动态数组的随机存取性体现不出多大的优势。随着元素的增加，动态数组的总尺寸需要根据情况做出相应的调整，而这种调整是一项很消耗时间的操作。从另一方面讲，如果安排得当，动态数组要比链表更有效率——因为链表必须为它的每一个元素进行动态内存分配。链表还必须为它的每一个元素准备一个指针，而这也要消耗一定的内存；要是你的堆栈只用来保存小尺寸的数据——比如整数，那么指针方面的额外开销将会占据一个显著的比例。考虑到上述这些因素，一个基于动态数组的堆栈其性能将大大优于一个基于链表的堆栈。但在程序设计面试中，你应该把注意力集中在你的解决方案是否简单易行以及能否很快地拿出一个让考官满意的结果。因为链表比动态数组更容易实现，所以你应该把链表做为你解决方案中的首选。

在向面试考官解释完你选择链表来实现堆栈的理由之后，你就可以开始设计有

关例程及其调用接口了。在动手编写代码之前多花点时间去完善你的设计方案是很有必要的，它不仅能降低你写出错误代码的几率，还能避免出现不同例程的调用接口彼此不统一规范的情况。更重要的是，这将显示出你具备有驾驭大型项目所不可缺少的一项能力——在动手编写代码之前先做出周密的计划。

push（压入）和pop（弹出）例程是每一种堆栈实现方案中都不可缺少的。那么，这两个例程的调用接口应该是怎样的呢？首先，你必须把某堆栈做为一个输入参数传递给这两个函数；其次，你还需要把将被压入堆栈的某项数据传递给push函数，而pop函数则必须把从堆栈中弹出的某项数据返回给它的调用者。要想把某个堆栈做为一个输入参数，最简单的办法莫过于在函数调用过程中传递一个指向该堆栈的指针。因为你的堆栈将被实现为一个链表，所以这个指向堆栈的指针就将是一个指向该链表第一个元素的指针。除这个指向堆栈的指针外，你还需要把那个将被压入堆栈的数据安排为push函数的第二个输入参数。pop函数有一个指向某堆栈的指针做为其输入参数就已经足够了，但它需要把从堆栈上弹出的数据值返回给它的调用者。为了最终确定这两个函数的调用模型，你还需要知道进出堆栈的数据是属于那种类型的；也就是说，你得声明一个struct结构做为链表元素的数据类型。如果面试考官没有对此做出要求，你就应该考虑使用无类型（即void）指针来实现一个通用的解决方案。基于无类型指针的struct结构和函数调用模型如下所示：

```
typedef struct elementT {
    struct elementT *next;
    void *data;
} element;

void Push(element *stack, void *data);
void* Pop(element *stack);
```

我们再从功能性和出错处理等方面出发来进一步考虑一下这两个例程。

函数push和pop都将对链表的头元素做出修改，而这些修改必须让它们的调用者“看”到才行。也就是说，你修改的必须是那个调用了push和pop的父函数中的堆栈指针。但在C语言里，你对传递给子函数做为其输入参数的某个指针所做出的修改是无法反映到其父函数里的。要解决这一问题，你就必须把一个指针的指针传递给被调用例程做为参数。也就是说，为了对其父函数中的堆栈指针做出修改，你必须把一个指向堆栈指针的指针传递给例程push和pop做为其输入参数；只有这样，你才能保证父函数中的堆栈指针正确地指向链表的头元素。考虑了这一因素的push和pop调用模型如下所示：

```
void Push(element **stack, void *data);
void* Pop(element **stack);
```

接下来，我们还要对出错处理做出安排。例程push需要为链表中的新元素动态地分配内存。内存分配是一项可能会失败的操作，所以你在编写push例程的时候必须要检查这个内存分配操作是否成功。你还需要把push操作是否成功的情况通知给它的调用者。在C语言里，利用子函数的返回值来表明它是否执行成功的做法最常见，也是最方便的。这样，我们就能在if语句的条件子句里对子函数进行调用，在if语句的语句体里对子函数的出错情况做出处理。因此，我们可以给push例程安排两个参数，一个是指向堆栈指针的指针，另一个是将被压入堆栈的数据；而它的返回值则以“true”表明执行成功，以“false”表明执行失败。

例程pop会出现执行失败的情况吗？它不需要进行动态内存分配，但如果出现对一个空堆栈进行弹出操作的情况该怎么办？在遇到这种情况的时候，它应该表明弹出操作没有成功；但在弹出操作执行成功的时候，它还需要把那个被弹出的数据值返回给它的调用者。我们知道，一个C函数只能有一个返回值，可pop函数却需要你为它安排两个返回值，一个是被弹出的数据，另一个是出错代码。

这一问题的解决办法有很多种，但没有一种能让人百分之百满意。一种方案是让一个返回值起两种作用：如果pop操作成功，它返回一个值；如果它操作不成功，则返回“NULL”。只要你的数据是指针类型且你永远不需要在堆栈里保存空指针，就可以考虑采用这一方案。但如果堆栈上可能出现空指针，你将很难判断pop的返回值到底是代表着一个合法的元素，还是表明该堆栈是一个空堆栈。在某些场合，规定堆栈里的指针不得为空也许是可行的，但就眼下这道面试题来说，我们不应该做出这样的假设。

如果不能采用只用一个返回值就分别起到数据和出错代码两方面的作用，你就必须明确地安排出两个不同目的的返回值。那么，除正常的返回值以外，还有什么办法能让函数再多返回一项或者几项数据呢？请大家再想想我们安排堆栈指针参数的做法：如果你传递给某函数的输入参数是一个指向某变量的指针，这个函数就能利用这个指针来修改那个变量的值并把有关数据返回给该函数的调用者。

如果你决定采用这个办法来返回两个值，pop函数的调用接口就将有两种设计方案。一种方案是给pop函数增加一个指向某出错代码变量的指针参数，让pop函数的返回值是被弹出的数据；另一种方案是给pop函数增加一个指向某数据变量的指针参数，让pop函数的返回值是出错代码。一般说来，大多数程序员倾向于让pop函数的返回值是被弹出数据的做法。但让pop函数的返回值不是出错代码的做法会带来其他一些不便：无法在if或while语句的条件子句里对pop函数进行调用——你将不得不明确地声明一个出错代码变量，并在程序代码里增加一条语句以便在每次调用完pop函数后对该变量的值进行检查。这将形成这样一种不规范的局面：push函数有一个数据性质的输入参数，返回的是一个出错代码；可pop函数却有一个出错代码性质的输

入参数，返回的是一项数据。这很容易把你自己给弄糊涂（至少我们是这样的）。我们刚讲过，大多数程序员倾向于让pop函数的返回值是被弹出数据的做法。公正地说，两种做法都不能令人百分之百地满意，它们都有着这样或那样的缺陷。但在程序设计面试中，你选择哪一种方案都行，因为你可以在现场对比分析两种方案的优劣并对你的选择做出解释。我们认为，以出错代码做为输入参数的方案要更差一些，所以我们下面的讨论将以你选择的方案是让pop函数返回一个出错代码的情况来继续。我们最终得到的是如下所示的调用模型：

```
int Push(element **stack, void *data);
int Pop(element **stack, void **data);
```

你也许还想编写两个分别用来创建和删除有关堆栈的CreateStack和DeleteStack函数。就基于链表的堆栈解决方案而言，这两个函数并不是必需的——如果你想删除某个堆栈，可以反复调用pop函数直到该堆栈变成一个空堆栈为止；如果你想创建一个堆栈，可以把一个指向“NULL”的指针传递给push函数做为它堆栈指针性质的输入参数。增加这两个函数的好处是，它们将使你的堆栈解决方案拥有一套既完备又与具体方案无关的程序设计接口。在增加了这两个函数之后，你的堆栈实现方案将更为完善——即便有人修改了堆栈本身的实现细节，使用了有关堆栈的程序也不需要修改。

既然我们的目标是实现出一种与具体方案无关且接口统一规范的堆栈，那么让CreateStack和DeleteStack函数也返回出错代码就将是一件顺理成章的事情了。虽说CreateStack和DeleteStack函数在基于链表的实现方案里永远都不会出现执行失败的情况，但它们在其他实现方案里却有可能不能成功（比如说，基于动态数组的实现方案里的CreateStack函数就有可能会在分配内存的时候遇到麻烦）。如果你没有在这两个函数里安排上表明它们执行失败的办法，当今后有人需要修改你的堆栈实现细节时，事情就会变得相当棘手。注意，我们将再次遇到与pop函数类似的麻烦——CreateStack函数也必须处理好其返回值代表的是一个空堆栈还是一个出错代码的问题。你不能使用“NULL”指针来表明执行失败，因为我们已经安排“NULL”指针在基于链表的堆栈实现方案里去代表一个空堆栈。为了保证整个方案的统一和规范，我们决定把CreateStack函数的调用接口实现为让它的返回值代表一个出错代码的样子。也就是说，CreateStack函数的返回值将不是那个指向新创建的堆栈的指针，因此，我们需要给它增加一个指向堆栈指针的指针做为其输入参数。同时，因为其他函数都有一个指向堆栈指针的指针做为输入参数，所以我们干脆也给DeleteStack函数也同样安排一个指向堆栈指针的指针做为输入参数好了。这样，如果你把堆栈指针声明为“element*stack”，那你永远可以用“&stack”来传递堆栈参数——你根本用不着去记忆哪些个函数的输入参数是“stack”，又有哪些个函数的输入参数是“&stack”。根据以上讨论，我们得到了如下所示的调用模型：

```
int CreateStack(element **stack);
int DeleteStack(element **stack);
```

在把所有这些事情都安排妥当之后，这几个函数的代码就很容易编写出来了。CreateStack函数将把堆栈指针设置为“NULL”，并返回一个表示操作成功的“1”。如下所示：

```
int CreateStack(element **stack)
{
    *stack = NULL;
    return 1;
}
```

push函数将为新元素分配内存，检查内存分配操作是否失败，对新元素进行赋值，把新元素放到堆栈的栈顶，最后对堆栈指针做出调整。如下所示：

```
int Push(element **stack, void *data)
{
    element *elem;
    elem = (element *) malloc(sizeof(element));
    if (!elem)
        return 0;
    elem->data = data;
    elem->next = *stack;
    *stack = elem;
    return 1;
}
```

pop函数先检查堆栈是否为空，然后取回栈顶元素中的数据值，对堆栈指针做出调整，再释放原栈顶元素（它此时已经不再属于这个堆栈了）所占用的内存。如下所示：

```
int Pop(element **stack, void **data)
{
    element *elem;
    if (!(elem = *stack))
        return 0;
    *data = elem->data;
    *stack = elem->next;
    free(elem);
    return 1;
}
```

DeleteStack函数可以用反复调用pop函数的办法来实现，但一边遍历堆栈数据结构（即构成堆栈的那个链表）一边释放各链表元素的办法将更有效率。在释放当前元素的时候，你需要使用一个临时指针来保存指向下一个元素的指针，这一点千万不要忘了。

```
int DeleteStack(element **stack)
{
    element *next;
    while (*stack) {
        next = (*stack)->next;
        free(*stack);
        *stack = next;
    }
    return 1;
}
```

在结束关于这道面试题的讨论之前，我们认为你还应该向面试考官说明这样一点：如果用C++等面向对象的程序设计语言来实现的话，各有关函数的调用接口将更加简单明了。CreateStack和DeleteStack函数相当于必要的构造器和解构器；而push和pop函数将被绑定在堆栈对象上，它们将不再明确地需要一个堆栈指针做为其输入参数，也就用不着采用“指针的指针”这种繁琐易错的安排了。更重要的是，当内存分配操作失败的时候，操作系统本身的内存管理功能将抛出一个例外来；这将使你能够用pop函数的返回值来返回被弹出的数据，而不是像现在这样返回一个出错代码。下面是我们用C++写出来的有关代码：

```
class Stack
{
public:
    Stack();
    ~Stack();
    void Push(void *data);
    void *Pop();
protected:
    // Element struct needed only internally
    typedef struct elementT {
        struct elementT *next;
        void *data;
    } element;

    element *firstEl;
};

Stack::Stack() {
    firstEl = NULL;
    return;
}

Stack::~Stack() {
    element *next;
    while (firstEl) {
        next = firstEl->next;
```

```
        delete firstEl;
        firstEl = next;
    }
    return;
}

void Stack::Push(void *data) {
    //Allocation error will throw exception
    element *element = new element;
    element->data = data;
    element->next = firstEl;
    firstEl = element;
    return;
}

void *Stack::Pop() {
    element *popElement = firstEl;
    void *data;

    /* Assume StackError exception class is defined elsewhere */
    if (firstEl == NULL)
        throw StackError(E_EMPTY);

    data = firstEl->data;
    firstEl = firstEl->next;
    delete popElement;
    return data;
}
```

这段C++代码要求那些使用着Stack类的应用程序必须具备用来完成出错处理的例外处理机制。但因为例外处理机制是一种推出时间并不很长的C++功能，所以这项要求并不是总能得到满足。如果你把我们用在C语言代码实现中的有关出错处理语句明确地添加到这段C++代码里来，就可以消除这段C++代码对例外处理机制的依赖。

## 3.5 面试例题：链表的尾指针

- **有一个单向链表，它的元素全都是些整数。head和tail分别是指向该链表第一个元素（即头元素）和最后一个元素（即尾元素）的全局性指针。请实现调用接口如下所示的两个C语言函数：**

```
int Delete(element *elem);
int InsertAfter(element *elem, int data);
```

**Delete函数只有一个输入参数，它就是那个将被删除的元素。InsertAfter函数有两个输入参数，第二个输入参数给出了新元素的取值，它将被插入到第一个输入参数所指定的元素的后面。当需要把新元素插入到链表的开头做为新的头元素时，函数**

**InsertAfter的第一个输入参数（即被声明为element类型的那个输入参数）将被设置为“NULL”。如果执行成功，这两个函数将返回“1”；如果不成功，将返回“0”。**

你编写出来的函数必须正确地调整好head和tail指针。

这道面试题看起来很简单。元素的删除和插入都是链表的常见操作，用头指针来定位链表的事情也大家所熟悉的。这道面试题惟一不同寻常的地方是它要求你必须正确地调整好head和tail指针，但这项要求并不会使链表的基本特性以及你在链表上的操作发生变化，所以你用不着去寻找什么新的算法。你只要在必要时对head和tail指针做出正确的修改就不会有问题了。

那么，什么时候需要对这些指针进行修改呢？很明显，发生在一个长链表中间部分的操作对head和tail指针是不会有什么影响的。只有当你的操作改变了位于链表首、尾两端的元素时，才需要对这两个指针做出调整。具体地说，如果在链表的某一端插入了一个新元素，那么被插入的那个元素就将成为链表新的头元素或尾元素；如果在链表的某一端删除了一个元素，那么原先的第二个或倒数第二个元素就将成为链表新的头元素或尾元素。

我们把发生在链表中间部分的操作归为普通情况，把发生在链表两端的操作归为特殊情况。有些特殊情况是很难考虑周全的，尤其是那些本身还有一些特例的特殊情况更是如此。排查特殊情况的技巧之一是先从有哪些因素可能会导致产生特殊情况来入手，再检查你预想的解决方案能否正确地应对这些因素；如果又出现了新问题，就说明你又发现了一个新的特殊情况。

除了在链表两端进行操作而产生的特殊情况外，“NULL”指针也是一个不容忽视的“意外”因素。既然操作发生在链表身上，就必然会对链表本身产生影响——尤其是它的长度。那么，链表的长度在什么条件下会导致特殊情况呢？一般说来，链表应该具备头、腹、尾三部分；如果某链表的这三部分不齐全，就肯定需要做一些特殊的考虑才能完成操作。首先，空链表连一个元素都没有，它的头、腹、尾三部分根本就无从谈起。其次，只包含一个元素的链表不具备“腹部”，那个惟一的元素将同时充当该链表的头元素和尾元素。再次，只包含两个元素的链表也不具备“腹部”，那两个元素将分别充当该链表的头元素和尾元素。当链表中的元素个数多于两个的时候，它的头、腹、尾三部分就将是齐全的；我们把这类链表归入普通情况——因为它们不太可能会导致特殊情况的出现。根据以上分析，当遇到长度为0、1、2个元素的链表时，必须采取一些特殊处理才能保证有关操作能够正确地完成。

对面试题本身的研究到这里告一段落，你现在可以开始编写Delete函数了。别忘了我们刚做的分析：链表头元素的删除操作需要特殊对待，所以你首先要把待删除元素的指针与head指针进行比较以决定是否需要进行特殊处理。如下所示：

```
int Delete(element *elem)
```

```
{
    if (elem == head) {
        head = elem->next;
        free(elem);
        return 1;
    }
    ...
```

接下来是对链表“腹部”元素进行删除的普通情况。你需要一个元素指针来记录链表元素的当前位置（我们称之为curPos指针）。还记得我们以前给出的如何在链表中删除一个元素的代码吗？你需要一个指向待删除元素前一元素的指针并修改其next指针。要想找出待删除元素的前一个元素，最简便的办法就是把“curPos->next”与“elem”元素进行比较；这样，当你找到“elem”元素时，curPos指针就将指向它的前一个元素。请认真编写你的遍历循环语句，不要遗漏掉某个元素。如果你把curPos指针初始化为指向链表头元素，那么“curPos->next”将指向链表的第二个元素。从第二个元素开始遍历循环是不会产生错误的，因为你已经把头元素做为一种特例处理过了——但一定要先进行比较操作后修改curPos指针，要不然就会遗漏掉链表的第二个元素。如果curPos指针变成了“NULL”，就说明你已经到达了链表的末尾，且没有找到你打算删除的那个元素——Delete函数的返回值应该表明这次删除操作没有成功。对链表“腹部”元素进行删除的操作将由以下代码完成（新增代码以黑体字显示）：

```
int Delete(element *elem)
{
    element *curPos = head;

    if (elem == head) {
        head = elem->next;
        free(elem);
        return 1;
    }

    while (curPos) {
        if (curPos->next == elem) {
            curPos->next = elem->next;
            free(elem);
            return 1;
        }
        curPos = curPos->next;
    }

    return 0;
    ...
```

接下来，完成对链表尾元素（即链表的最后一个元素）的删除操作。链表尾元

素的next指针是“NULL”。要想删除链表的尾元素，你需要把链表倒数第二个元素的next指针修改为“NULL”并释放尾元素占用的内存。仔细观察一下对链表“腹部”元素进行删除操作的代码就会发现，它完全能够正确地删除链表的尾元素——就像对待某个位于链表“腹部”的元素一样。惟一需要注意的地方是：如果你删除的是尾元素，那你还需要对tail指针做相应的修改。当把“curPos->next”设置为“NULL”时，你应该知道自己已经改变了链表的尾元素，这就要求你必须对tail指针也做出相应的修改。把这一点添加到你的函数里，你将得到如下所示的代码：

```
int Delete(element *elem)
{
    element *curPos = head;

    if (elem == head) {
        head = elem->next;
        free(elem);
    }

    while (curPos) {
        if (curPos->next == elem) {
            curPos->next = elem->next;
            free(elem);
            if (curPos->next == NULL)
                tail = curPos;
            return 1;
        }
        curPos = curPos->next;
    }

    return 0;
}
```

这个解决方案覆盖了我们前面讨论过的、与待删除链表元素位置有关的三种特殊情况。在把这个解决方案提交给面试考官之前，你还需要检查你的Delete函数在以“NULL”指针做为输入参数以及在与链表长度有关的三种特殊情况中的操作情况。当输入参数“elem”的调用值是“NULL”时会发生什么情况呢？请看代码：while循环将遍历链表直到“curPos->next”等于“NULL”为止（此时，curPos指针将指向链表的尾元素）；但此时下一条语句中的“elem->next”将对一个“NULL”指针进行操作。从链表中删除“NULL”是一项不可能的操作，所以这里就产生了一个问题；解决这一问题最简单的办法是让Delete函数在elem等于“NULL”时返回一个“0”。

如果链表里有零个元素，那么head和tail指针都将是“NULL”。此时，Delete函数最开始处的“if (elem == head)”就相当于“if (elem == NULL)”，求值结果将永远

为假。同时，因为head指针是“NULL”，所以curPos指针也将是“NULL”，while循环的循环体将永远不会被执行到。这样看来，零元素链表将不存在任何问题——Delete函数将返回一个“0”，因为空链表里没有可删除的东西。

再来看看只有一个元素的链表。此时，head和tail指针将都指向那个惟一的元素——它也是你惟一可删除的元素。当“if （elem==head）”为真时，“elem->next”肯定等于“NULL”，于是你的Delete函数将正确地把head指针设置为“NULL”并释放这个元素；但tail指针却仍指向着你已经释放了的这个元素。也就是说，对于只有一个元素的链表，你还需要特意把tail指针设置为“NULL”才算是正确、圆满地完成了元素删除操作。有且只有两个元素的链表又会怎么样呢？删除第一个元素将使head指针指向第二个元素，没有问题；删除第二个元素将使tail指针指向第一个元素，也没有问题。也就是说，缺少“腹部”元素并不会产生什么意外。现在，你只要在Delete函数里再添加两种特殊情况就可以去研究InsertAfter函数的编写方案了：

```
int Delete(element *elem)
{
    element *curPos = head;

    if (!elem)
        return 0;

    if (elem == head) {
        head = elem->next;
        free(elem);
        /* special case for 1 element list */
        if (!head)
            tail = NULL;
        return 1;
    }

    while (curPos) {
        if (curPos->next == elem) {
            curPos->next = elem->next;
            free(elem);
            if (curPos->next == NULL)
                tail = curPos;
            return 1;
        }
        curPos = curPos->next;
    }

    return 0;
}
```

在编写InsertAfter函数的时候，需要考虑的因素与我们前面的分析大同小异。因

为这个函数需要为新元素分配内存，所以千万不要忘记检查内存分配操作是否成功以避免出现内存泄漏。Delete函数中的大部分特殊情况都需要在InsertAfter函数中加以注意，这两个函数在整体结构上非常相似：

```
int InsertAfter (element *elem, int data)
{
    element *newElem, *curPos = head;

    newElem = (element *) malloc(sizeof(element));
    if (!newElem)
        return 0;
    newElem->data = data;

    /* Insert at beginning of list */
    if (!elem) {
        newElem->next = head;
        head = newElem;

        /* Special case for empty list */
        if (!tail)
            tail = newElem;
        return 1;
    }

    while (curPos) {
        if (curPos == elem) {
            newElem->next = curPos->next;
            curPos->next = newElem;

            /* Special case for inserting at end of list */
            if (!(newElem->next))
                tail = newElem;
            return 1;
        }
        curPos = curPos->next;
    }

    /* Insert position not found; free element and return failure */
    free(newElem);
    return 0;
}
```

这道面试题重点考察了求职者对特殊情况的分析和处理能力。虽说不怎么有趣，也不太容易做到百分之百的满意，但它确实是一道非常有针对性的面试题。面试考题多少都会包含有一些特殊情况，所以你必须有随时会遇到它们的思想准备。在实际编程工作中，未做适当处理的特殊情况就是程序代码中的bug（程序漏洞），而且

还是那种很难查找、重现、和修正的bug。一般说来，一位在编写代码的同时就已经考虑到各种特殊情况的程序员肯定要比那些只知通过调试去查找bug的程序员来的高明，也更有效率。经验丰富的面试考官肯定都知道这一道理，也肯定会在面试过程中注意求职者们的表现，看他们是主动地把特殊情况的处理当做代码编写工作的一部分，还是被动地在经过提示之后才觉察到面试题中隐藏着的特殊情况。

## 3.6 面试例题：对RemoveHead函数进行纠错

- **下面是一个用来删除单向链表的头元素的函数。请找出其中的程序漏洞并加以纠正。**

```
void RemoveHead(node *head)
{
    free(head);          /* Line 1 */
    head = head->next;   /* Line 2 */
}
```

这种调试纠错类的考题经常出现在程序设计面试中，所以多花点时间来研究一套对付这类考题的通用策略是非常有意义的。

在参加程序设计面试的时候，供你进行分析的代码往往只有很短的一小段，所以你的调试纠错策略与实际程序设计工作中所使用的也应该有所不同。虽说用不着顾虑面试题中的代码与其他程序模块的接口以及它们之间相互影响等问题，但你绝不能掉以轻心；你必须在不借助调试器的情况下对有关代码的每一行做认真、系统的分析。在拿到面试考官给出的函数或代码后，你应该从以下四个问题比较集中的方面进行检查和分析：

- **进入函数的数据是否都“来历”清白，类型无误。**

函数中是否有未被声明的变量？被当做（比如说）“int”类型使用的数据是不是“long”类型？完成操作所必需的数据是否齐全？

- **函数中的每一条语句是否执行正确、正常。**

函数能否完成预定的功能？语句的执行顺序是否正确？各语句本身有无错误和漏洞？能否在代码段的末尾产生预想的结果？

- **离开函数的数据是否都“去向”明确，格式无误。**

返回值是不是人们所预期的？如果子函数需要改变其调用者（即父函数）中的某个变量，那它是否正确地做到了这一点？

- **函数对常见的特例是否进行了处理。**

不同的问题有不同的特例，它们通常与不寻常的输入参数有关。比如说，对某种数据结构进行处理的函数在遇到空白或者几乎空白的数据项时往往会产生意外结果；以指针做为输入参数的函数在该指针是“NULL”时往往会无法执行失败；等等。

调试纠错工作将从第一步——检查进入函数的数据是否都“来历”清白且类型无误——开始。就我们这道关于链表的面试题而言，你只要知道了它的头指针，就能对它的每一个结点进行访问。因为RemoveHead函数的输入参数确实是一个链表的头指针，所以你肯定能通过它来找到你需要的数据——这一步没发现错误。

接下来，对有关语句逐条进行分析。第1行释放了头元素——暂时没有问题。第2行给head指针赋了一个新值，可它是用head指针的旧值完成的。这可是一个错误：既然已经释放了head，你又怎么能引用它呢？我们现在把第1和第2行语句的顺序颠倒一下试试，没能解决问题——这将导致head后面的那个元素被释放。你需要释放head，但在释放之后你还需要用到它的next指针值。你可以用下面这个办法来纠正这个毛病：用一个临时指针变量来保存head元素的next指针值，然后释放head，再用那个临时变量来修改头指针head。按这个办法修改后的RemoveHead函数如下所示：

```
void RemoveHead(node *head)
{
    node *temp = head->next;  /* Line 1 */
    free(head);               /* Line 2 */
    head = temp;              /* Line 3 */
}
```

下面，我们将进入第三个分析步骤——检查RemoveHead函数的返回值是否正确。这个函数没有给出一个明确的返回值，但它有一个隐含的返回值——这个函数的用途是修改其调用者（父函数）里的head指针值。在C语言里，输入参数都是以值传递（pass by value）的方式进入子函数的，换句话说，子函数所使用的是输入参数的一个局部副本，对这个局部副本的修改不可能在该函数以外的地方被“看到”。也就是说，RemoveHead函数第3条语句对head的赋值操作根本达不到预想的目的——我们又发现了一个错误。要想纠正这个错误，我们必须想出一个能够改变父函数代码中的head指针值的办法。在C语言里，变量不能以引用传递方式（pass by reference）进入子函数，但你可以采用把一个指向变量的指针做为子函数输入参数的办法来绕开这个限制，就这道面试题来说，我们需要把一个指向head指针的指针传递给RemoveHead函数做输入参数。如下所示：

```
void RemoveHead(node **head)
{
    node *temp = (*head)->next;  /* Line 1 */
    free(*head);                 /* Line 2 */
    *head = temp;                /* Line 3 */
}
```

现在到达第四步——检查RemoveHead函数是否对特例进行了适当的处理。与这道面试题有关的特例有两种，一种是仅有一个元素的链表，另一种是空白链表。经过

分析，这个函数在遇到仅有一个元素的链表时不会出问题：它在删除掉那个惟一的元素之后会把head设置为“NULL”，表示原来的头元素已经被删除掉了。现在来分析空白链表的情况。空白链表其实就是一个“NULL”指针。如果head是一个“NULL”指针，那么上面这段代码中的第一条语句将对一个“NULL”指针进行操作——我们又找出了一个错误。纠正这一错误的办法并不难想到：在进入RemoveHead函数后，首先检查head是不是一个“NULL”指针；如果是，就说明链表是空白的，不需要进行指针操作，直接返回即可。增加了这一项检查的RemoveHead函数如下所示：

```
void RemoveHead(node **head)
{
    node *temp;
    if (!(*head)) {
        temp = (*head)->next;
        free(*head);
        *head = temp;
    }
}
```

好了，经过上面这些调试和纠错之后，我们可以确定RemoveHead函数（1）能够完成预定工作；（2）能够被正确地调用，也能够产生正确的返回值；（3）能够对有关特例做出适当的处理。你现在可以向面试考官报告说你已经完成了调试工作，并把我们最后得到RemoveHead函数代码做为你的答卷提交给考官们审查了。

## 3.7 面试例题：链表中的倒数第*m*个元素

- **给定一个单向链表，请设计一个既节省时间又节省空间的算法来找出该链表中的倒数第*m*个元素。实现这个算法，并为可能出现的特例情况安排好处理措施。“倒数第*m*个元素”是这样规定的：当 *m*=0 时，链表的最后一个元素（尾元素）将被返回。**

这道面试题有相当的难度，为什么这么说呢？单向链表是一种只能沿从头至尾的方向进行遍历的数据结构，如果这道题让我们寻找的是链表中的正数第*m*个元素，事情就简单到极点了。可这道题却偏让你去寻找从链表尾算起的倒数第*m*个元素。它的难度在于：在对链表进行遍历的时候，你不知道它的末尾在上面地方；而等你到达链表尾时，又很难回溯*m*个元素并找到你想要的东西。

在需要频繁查找倒数第*m*个元素的场合，你肯定不会选用单向链表做为你解决方案中的数据结构，因为它根本不适合用来完成这项任务。如果在实际编程工作中遇到类似的问题，你肯定会毫不迟疑地选择一种更适用的数据结构（比如一个双向链表）来代替单向链表。这些想法该不该说给面试考官们听呢？应该，这体现出你对“程序设计”这个词的含义有着深入的理解。可说归说，做归做；你还是应该遵照面

试考官开列出来的条件来解答这道面试题。

既然这种数据结构无法沿逆向进行遍历，你还能想出什么办法来解决这个问题呢？你想找的元素是从链表尾开始算起的倒数第$m$个元素；如果沿从头至尾的方向从链表中的某个元素开始遍历$m$个元素后刚好到达链表尾，那么该元素就是你要找的东西。这就引出了第一种办法：对链表中的每一个元素进行这样的测试，直到你找到那个倒数第$m$个元素为止。很明显，这个方案的效率是很差的，因为你将对同一批元素进行反复多次的遍历。仔细研究一下这个方案就会发现，对于链表中的大部分元素，你都要遍历$m$个元素；如果这个链表的长度是$n$个元素，那这个算法的执行时间大约是O($mn$)级的。你应该尝试寻找一个优于O($mn$)级的解决方案。

如果在对链表进行遍历的同时把一些元素（或者，指向这些元素的指针）存储起来会怎么样？这样，当你到达链表尾时，就可以在你存储起来的数据结构中回溯$m$个数据项并找出你想要的链表元素。如果你选用了适当的临时存储数据结构，这个算法就将是O($n$)级的，因为你只需对链表做一次遍历就能达到目的。不过，这个方案也不那么完美：如果$m$的取值很大，你的临时存储区也将会很大；在最坏的情况下，这个方案所需要的临时存储空间将与链表本身所占用的空间差不多大——这一方案肯定不是最节约空间的算法。

把焦点集中在链表尾也许并不是解决这道题目的最佳思路。既然从链表头开始的遍历是小菜一碟，那么有没有一种办法能让我们从链表头入手来找出想要的东西呢？我们的目标是找出从链表尾算起的倒数第$m$个元素，而这个$m$的值是我们已经知道了的。我们不妨假设这个元素是从链表头算起的第$x$个元素——虽然我们不知道这个$l$的值到底是多少，但它肯定满足等式“$x + m = n$”，其中$n$为链表的总长度。链表中有多少个元素是很容易被统计出来的，然后，计算出“$x = n - m$”，再从链表头开始遍历$x$个元素，不就找到想要的东西了吗？虽说整个过程需要对链表做两次遍历，但它仍是一个O($n$)级的算法，而且它只需要很少的几个临时变量。相对于此前的方案，这一思路的时间和空间效率有了显著的改进。要是你能设法省下统计链表长度的那一次遍历，这个算法就更有效率了。在这里提醒大家一句，根据题目要求，面试考官很可能希望你找出一个不需要增加或者修改数据结构的解决方案，如果你就此事询问面试考官，他很可能会明确地加上一些链表访问方法方面的限制。

如果你不能省略统计连元素个数的那次遍历，那你肯定要对链表做两次遍历。如果这是一个很长的链表而计算机的内存容量又很有限，那么这个链表的大部分很可能都存放在切换出物理内存的虚拟内存上（即存放在磁盘上）。如果真是这样，那么链表的每次遍历都将需要做大量的磁盘读写操作才能把链表的有关部分读到物理内存里来进行处理。在这种情况下，如果你的算法只需对链表做一次完整的遍历，那它将比必须做两次遍历的算法快很多——虽然它们都是O($n$)级的算法。那么，有没

有办法只通过一次遍历就找到那个目标元素呢？

很明显，从头开始统计链表元素个数的算法要求你确知链表的长度。如果你不知道链表的长度，就只能通过完整地遍历链表的办法来统计之。照这么看，似乎很难把这个算法的遍历次数压缩到只需进行一次的地步了，对吗？不对，我们再来研究一下刚才那个只需进行一次遍历但需要大量临时存储空间的算法。有没有可能把这个算法的空间开销压缩下来呢？

当到达链表尾的时候，你真正感兴趣的元素其实只有一个，它就是你一直想找的那个倒数第$m$个元素——也就是从你当前位置（因为你此时已经到达了链表尾）算起的倒数第$m$个元素。你之所以要回溯$m$个元素，是因为只有这样才能确定倒数第$m$个元素的位置，而这个位置是随着你遍历链表时的当前位置而变化的。如果你安排一个队列，让它的长度是$m$个元素，然后把遍历单向链表时的当前元素插入到这个队列头，移动单向链表遍历指针时再从这个队列尾删除一个元素，就能保证这个队列的头元素永远是你遍历单向链表时从当前位置倒数的第$m$个元素。但我们想再次提醒大家一句，面试考官很可能希望你找出一个不需要增加或者修改数据结构的解决方案，如果你就此事询问面试考官，他很可能会明确地加上一些链表访问方法方面的限制。

仔细思考一下就会明白，之所以需要这个$m$个元素长的队列，目的不过是想利用这种隐含的办法来获得一种能够与单向链表的遍历指针按一定间隔同步前进的办法而已。再仔细思考一下就会发现，这个队列其实并没有存在的必要——你完全可以明确地再安排一个用来遍历单向链表的指针，让它与链表的第一个遍历指针间隔$m$个元素，然后让两个指针同步前进。这个办法要比采用一个$m$个元素长的队列来隐含地达到这一目的更简便，因为你彻底省略了队列上的元素插入和删除操作。这个算法好像就是我们一直在寻找的东西：线性的执行时间，只需对链表做一次遍历，微不足道的存储空间要求。你只要再把细节部分考虑好就可以交卷了。

你需要用到两个指针，为了便于区别，我们把最先出发的链表遍历指针称为“当前指针”，把另一个稍后出发的与当前指针相距$m$个元素的指针称为“拖后指针”。你必须保证这两个指针之间的间隔是$m$个元素，然后再让它们同步前进。当你的当前指针到达链表尾的时候，拖后指针将恰好指向从链表尾算起的倒数第$m$个元素。怎样才能让这两个指针保持正确的间隔呢？你可以这样做：从当前指针从链表头出发时开始统计链表元素的个数（注意要从“0”开始计算），等数到第$m$个的时候，再让拖后指针从链表头开始出发；这就能保证两个指针的间隔是$m$个元素。在这一过程中，有没有需要特殊处理的特殊情况呢？有，如果链表的长度不足$m$个元素，它就不存在倒数第$m$个元素。此时，如果你试图让当前指针前进$m$个元素，就很可能会在半路上遇到对“NULL”指针进行操作的情况。因此，你必须在这两个指针的出发阶段检查最先出发的当前指针是否已经到达了链表尾。

有了这一考虑之后，你就可以动手编写这个算法的代码了。大家要注意的是，在需要精确间隔$m$个元素或者需要准确到达第$m$个元素位置的场合，稍不小心就会在编写代码时弄出前后错位一个元素的错误来。你必须根据面试题对“倒数第$m$个元素”的准确定义多试验几个例子，确保自己不会数错元素个数——尤其是在最先出发的当前指针（即代码中的current指针）的出发阶段。

```
element *FindMToLastElement(element *head, int m)
{
    element *current, *mBehind;
    int i;

    /* Advance current m elements from beginning,
     * checking for the end of the list
     */
    current = head;
    for (i = 0; i < m; i++) {
        if (current->next) {
            current = current->next;
        } else {
            return NULL;
        }
    }

    /* Start mBehind at beginning and advance pointers
     * together until current hits last element
     */
    mBehind = head;
    while (current->next) {
        current = current->next;
        mBehind = mBehind->next;
    }

    /* mBehind now points to the element we were
     * searching for, so return it
     */
    return mBehind;
}
```

## 3.8 面试例题：链表的扁平化

- **给定一个双向链表。这个双向链表中的每一个元素除固有的后指针（指向后一个元素）和前指针（指向前一个元素）外，还有一个子指针，每个子指针可能指向也可能不指向另一个双向链表。而那些子双向链表本身还可能有一个或者多个子双向链表，从而形成一种多层次的数据结构，如图3-3所示。**

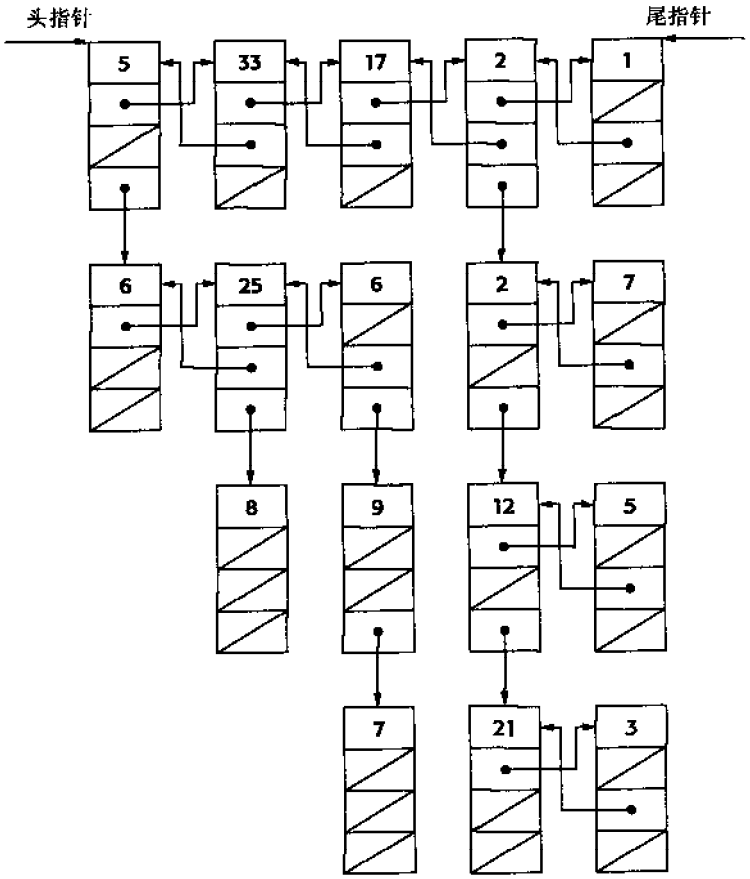



图3-3 由双向链表构成的多层次数据结构

**对这个链表进行扁平化，使全体结点都出现在一个只有一个层次的双向链表里。已知条件只有原多层次双向链表的第一层次的头指针和尾指针。下面是各结点的C语言struct定义：**

```
typedef struct nodeT {
    struct nodeT *next;
    struct nodeT *prev;
    struct nodeT *child;
    int value;
} node;
```

这道链表扁平化试题有很大的自由发挥空间，它只要求你对链表进行扁平化而没有限制具体的做法。这项任务可以用很多种办法去完成，它们都能让你得到一个只有一个层次的双向链表，只是链表结点的排列顺序会各有不同而已。

我们准备先向大家介绍几种不同的算法以及它们各自生成的结点顺序，然后再挑出一种最容易实现、效率也最高的算法来加以实现。

先来研究一下这种数据结构本身。从链表的角度看，这是一种不多见的数据结构。它有层次和子链表——一倒有点像是一棵树（tree）。树这种数据结构也具备层次和子结点，但位于同一层次的树上结点彼此是不相连的。如果追求简单的话，你只需随便选用一种常见的树遍历算法并在遍历过程中把各个结点复制到一个新的链表里去就可以完成对这种数据结构进行扁平化的任务了。

但这种数据结构毕竟不同于我们所熟悉的树，所以你必须对你所选用的树遍历算法加以修改才能对它进行遍历处理。如果把这种数据结构看做是一棵扩展了的树，那它上面的每一个子链表就相当于一个扩展了的树结点。这看起来并不坏：标准的树遍历算法将直接检查这棵扩展树上每一个树结点的子指针（*child），你只需用一种链表遍历算法来检查所有的子指针就行了；每遇到一个新的结点，就把它拷贝到一个复制品链表里去，而这个复制品链表就将是你的扁平化链表。但在琢磨这个解决方案的细节之前，我们还应该先来看看它的执行效率。

每个结点都将被遍历一次，所以这个算法将是一个O($n$)级的解决方案。遍历操作会增加一些递归或数据结构方面的开销。此外，你还需要把每个结点都拷贝到最终的扁平化链表里去。但这种拷贝操作是一种效率比较低下的活动，尤其是当这个结构的尺寸比较巨大的时候。因此，我们建议你先不要动手去编写代码，看自己还能不能找到一种不需要做这么多拷贝操作的高效率算法。

前面提到的解决方案把重点放在了算法上，没有考虑结点排列顺序方面的因素。现在，我们将把注意力集中到结点的排列顺序上，并根据我们的观察和研究推导出一个算法来。我们将从该数据结构的层次关系开始我们对其结点排列顺序的研究。为了便于讨论，我们将把该数据结构的一个层次简称为“子链表”（child list）。就像旅馆中的房间是按楼层来排列其顺序那样，你也可以按结点所在的层次来排列其顺序。每个结点都有它自己的层次，而出现在同一层次上的结点可以按某种顺序排列起来（比如按从左至右的出现次序排列）。这样，这些结点就能形成一种类似于旅馆房间号码的逻辑排列顺序。你可以先排列所有出现在第一层次的结点，然后是所有出现在第二层次的结点，再往后是所有出现在第三层次的结点，依次类推。对面试题中给出的示例图做如此排列后将得到如图3-4所示的顺序。

现在，我们将试着根据这个顺序来总结出一个算法。这个顺序的特点之一是你用不着对出现在同一层次的结点再进行排序。你只要把出现在同一层次上的结点串联起来，再把不同层次串联在一起就能完成这道面试题的解答。但要想把出现在同一层次上的结点串联起来，你必须先把它们一个不漏地都找出来，这就要求你必须在该层次上先做一次宽度优先搜索（breadth-first search）。但宽度优先搜索的执行效

率并不高，所以你还应该继续努力，看能不能再找出一个更好的解决方案来。

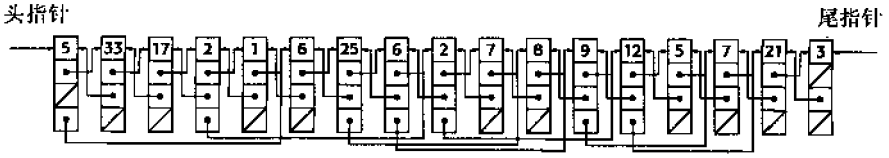



图3-4 经过排序的链表结点

在图3-3里，第二层是由两个子链表构成的。每个子链表都以位于第一层的某个结点的子结点开始。你可以考虑把这两个子链表依次追加链接到第一层的末尾去——如此一来，你就用不着合并这两个子链表了。

依次追加链接子链表的工作可以按以下步骤进行：从第一层的头元素开始沿各结点的next指针进行遍历；每遇到一个子结点，就把子结点（包括相应的子链表）追加链接到第一层的末尾并对tail指针做出相应的修改。你迟早会把第二层完整地追加到第一层的末尾去。继续对第一层进行遍历，但你遇到的将是那些原属于第二层的结点。不断重复把子链表追加链接到第一层的上述步骤，你迟早会把所有的子链表都追加完毕，并得到一个符合要求的扁平化链表。这个算法可以被表述为如下所示的形式：

```
从第一层的头元素开始
当你尚未到达第一层的末尾时
    如果当前结点有子结点
        把子链表追加链接到第一层的末尾
        对tail指针做相应的修改
    前进到下一个结点
```

这个算法非常简单，所以很容易实现。从执行效率的角度看，第一层之后的每个结点都将被遍历两次，一次是为了在追加链接了子链表后对tail指针做相应的修改，另一次是为了查看它们本身还有没有子结点。第一层的结点只在你检查它们有没有子结点时被遍历了一次，因为最初的tail指针（即第一层的tail指针）是已知的。也就是说，这个算法最多会进行2n次比较操作，是一个O($n$)级的解决方案。这已经是你所能获得的最佳执行时间规模了，因为你必须对每一个结点都进行检查。（这道面试题还有其他几种执行效率相当的解决方案，比如把子链表直接插入到它的父结点之后而不是追加链接到第一层的末尾。）

这个算法的实现代码如下所示（为了在函数返回时保留tail指针在本函数里的修改效果，你需要一个指向tail指针的指针——即**tail——来做为本函数的输入参数）。

```
void FlattenList(node *head, node **tail)[3]
{
    node *curNode =  head;
    while (curNode){
        /* The current node has a child */
        if (curNode->child) {
            Append(curNode->child, tail);
        }
        curNode = curNode->next;
    }
}

/* Appends the child list to the end of the tail and updates
 * the tail.
 */
void Append(node *child, node **tail)
{
    node *curNode;

    /* Append the child child list to the end */
    (*tail)->next = child;
    child->prev = *tail;

    /*Find the new tail, which is the end of the child child
     *list.
     */
    for (curNode = child; curNode->next;
        curNode = curNode->next)
        ; /* Body intentionally empty */

    /* Update the tail pointer now that curNode is the new
     * tail.
     */
    *tail = curNode;
}
```

**• 对链表进行反扁平化。把数据结构恢复成它进入FlattenList函数之前的原貌。**

你对这种数据结构应该是很熟悉的了。你已经知道如何把它的子链表合并在一起以构成一个扁平化链表。现在，我们需要把一个长长的扁平化链表恢复为它原先具有多层次子链表的样子。首先，我们可以依照用来创建扁平化链表的算法反其道而行之。在对链表做扁平化的时候，你是从第一层的头元素开始进行遍历并依次把各子链表追加链接到第一层末尾的。按照反其道而行之的原理，你需要从tail指针开始做逆向遍历并对扁平化链表进行分断，当遇到是一个子链表头元素的结点时，从这个结点开始把子链表从扁平化链表里分离出来即可。只可惜这个办法说着容易，做起来却很困难，因为没有上面好办法能让你用来判断某个结点是不是原数据结构中的一个子结点（即

某个子链表的头元素）。判断某结点是不是一个子结点的惟一办法是对出现在它前面的所有结点进行遍历扫描并检查它们的子指针。而这种扫描工作的效率是很低的，所以你应该开动脑筋，看还能否找到其他一些更具执行效率的解决方案来。

既然想避开子结点难以判断的困难，从链表的头元素入手当然是最容易想到的。你可以从头元素开始，把所有指向子结点的指针另外保存到一个数据结构里，然后再对链表进行回溯并分离出各子结点来。这个方案的好处是你用不着为了判断各个结点是不是一个子结点而反复多次地扫描链表。这是一个很不错的解决方案，但它需要一些额外的数据结构才能实现。能不能找到一个不需要额外数据结构的解决方案呢？

既然对链表做逆向遍历的想法已经山穷水尽，我们不妨找个按正常顺序对链表进行遍历的算法来试试。当然，你还是无法迅速判断出某个结点是不是一个子结点。按正常顺序进行遍历的好处是你将按对原链表做扁平化处理的顺序——也就是追加链接那些子链表的顺序——遭遇到那些子结点，而你遭遇到的每一个子结点都是原链表中的某个子链表的头元素。如果把子结点从它前面的那个结点处分断开，你就能得到扁平化之前的链表。

注意，你不能简单地从头开始遍历链表并在遇到一个有子结点的结点时就把该子结点与它前面的那个结点分断开。你应该直接前进到原先的第一层与第二层之间的连接点位置，然后在这个位置把链表分断开来；此时，你已经把原数据结构中的第一层和第二层分断出来了，但原数据结构中的其他结点还没有被遍历到。这个方案好像很不错。接下来，从新分断出来的子链表第一层（即子链表本身）的头元素开始进行遍历；当你又遇到一个有子结点的结点时，像刚才那样进行分断（分断出原数据结构的第二层和第三层），然后再对新分断出来的子链表进行遍历。等把子链表都遍历穷尽后，再继续遍历倒数第二层子链表；如此循环，直到完成整个任务。你可以把这些断点位置保存到一个数据结构里供稍后使用。不过，与其不怕麻烦地设计和实现这样一个数据结构，还不如直接采用递归技术。具体地说，每遇到一个有子结点的结点，就把相应的子链表分离出来，然后先遍历新分离出来的子链表，再继续遍历原来的子链表。

这是一个很有执行效率的算法，因为每个结点最多会被遍历两次，故执行时间将是O($n$)级。就这道面试题来说，O($n$)级的执行时间是已经是最好的了——为了检查某个结点是不是一个子结点，你至少要遍历它一次。在平均情况下，调用递归函数的次数将比结点的总个数少很多，所以因递归而增加的开销还是合算的。即使在最坏情况下，调用递归函数的次数也不会多于结点的总个数。这个递归方案的执行效率与刚才提到的那个需要额外数据结构的方案不相上下，程序代码却更简明易写。因此，这个递归算法应该是这道程序设计面试题的最佳解决方案了，我们可以把它表述为如下所示的形式：

从链表的头元素开始进行遍历：

```
当你尚未到达链表尾时
    如果当前结点有子结点
        把相应的子链表与前一个结点分断开
        从子链表的头元素开始进行遍历和分断
    前进到下一个结点
```

这个算法的代码实现如下所示：

```
/*This is a wrapper function that also updates the tail pointer.*/
void Unflatten(node *start, node **tail)
{
    node *curNode;
    ExploreAndSeparate(start);

    /* Update the tail pointer */
    for (curNode = start; curNode->next;
                curNode = curNode->next)
        ; /* Body intentionally empty */
    *tail = curNode;
}

/* This is the function that actually does the recursion and
 * the separation
 */
void ExploreAndSeparate(node *childListStart)
{
    node *curNode = childListStart;

    while (curNode) {
        if (curNode->child) {
            /* terminates the child list before the child */
            curNode->child->prev->next = NULL;
            /* starts the child list beginning with the child */
            curNode->child->prev = NULL;
            ExploreAndSeparate(curNode->child);
        }
        curNode = curNode->next;
    }
}
```

## 3.9 面试例题：空链表与循环链表

- 给定一个链表，它可能是一个以“NULL”结尾的非循环链表，如图3-5所示；也可能是一个循环结构结尾的循环链表，如图3-6所示。

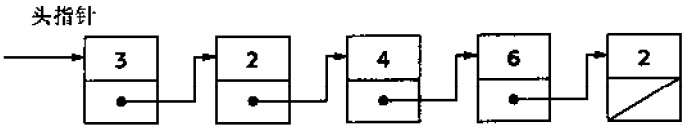



图3-5 非循环链表

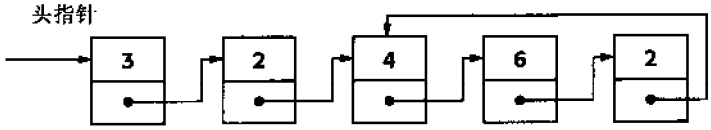



图3-6 循环链表

**已知这个链表的头指针，请编写一个函数来判断该链表是一个循环链表还是一个非循环链表，该函数不得对链表本身做任何修改。**

请仔细观察上面这两个示意图，看能不能发现循环链表和非循环链表之间可供利用的差异。

两种链表的主要差异在于它们的结尾部分：在循环链表的末尾，有一个结点的next指针指向的是出现在它前面的某个结点；在非循环链表的末尾，有一个结点的next指针指向的是“NULL”。如果你能把这个结点找出来，就可以查出链表是循环的还是非循环的了。非循环链表的尾结点是很容易判断的——对它的遍历操作迟早回遇到一个以“NULL”终结的结点。但循环链表的尾结点可就没那么好找了——你很难判断自己是仍在遍历一个很长的非循环链表，还是在一个循环链表的末尾部分兜圈子。要想解答出这道试题，你还得另辟奚径。

请再仔细看看循环链表的尾结点。看出什么了吗？这个尾结点的后续结点也是另一个结点的后续结点；换句话说，有两个指针是指向同一个结点的，这个结点也是链表中同时被两个元素所指向着的惟一结点。你可以根据这个特性设计出一个算法，让它在遍历链表并检查每一个元素，看是否有两个指针同时在指向着它。如果你找到了一个这样的结点，就说明该链表是循环的；否则，该链表就是非循环的——你迟早会遇到一个“NULL”指针。

可惜的是，指向某元素的指针到底有几个是很难查出来的。你发现循环链表结尾部分的其他特点了吗？在对循环链表进行遍历的时候，尾结点的下一个结点将是你此前曾经遍历过的。你可以放弃检查某结点是否被两个指针所指向的想法，转而去判断它是不是已经被你遍历过了。如果你找到了一个曾经被遍历过了的结点，就说明那是一个循环链表；如果你最终遇到了一个“NULL”指针，就说明那是一个非循环链表。这个算法的雏形已经出来了，你还需要找出一个判断某结点是否已被遍历过的办法来。

最简单的办法是给每一个被遍历过的元素加上一个标记，可这道考题的要求是你不得对链表做任何修改。记录链表结点是否已被遍历过的办法是有的，比如说，你可以每遍历一个结点，就把它记录到一个“已遍历结点清单”里去，然后再把当前结点与“已遍历结点清单”里的各个结点进行比较。如果当前结点的后续结点已经出现在了“已遍历结点清单”里，就说明那是一个循环链表；否则，你迟早会到达链表的末尾并遇到一个“NULL”指针，说明那是一个非循环链表。这个办法不是不行，但在最坏情况下，“已遍历结点清单”所需要的存储空间将与原始链表的一样大。还能不能把这个内存开销压缩一些呢？

你保存到“已遍历结点清单”里去的信息是什么样的呢？“已遍历结点清单”中的第一项对应于原始链表的第一个结点，第二项对应于原始链表的第二个结点，第三项对应于原始链表的第三个结点……这不等于是在为原始链表做一个镜像复制品吗？这可没有必要——你完全可以直接使用原始链表嘛。

在进行遍历的时候，你知道你的当前结点肯定在链表里，也知道这个链表的头指针，所以你完全能够把当前结点的next指针与它前面所有结点的next指针进行比较。如果你正位于第$m$个结点，就需要把它的next指针与头结点到第$m-1$个结点的next指针进行比较，如果出现两个next指针相等的情况，就说明那是一个循环链表。

这个算法的执行时间是多少？在第一个结点上，你需要进行0次比较；在第二个结点上，你需要进行1次比较；在第三个结点上，你需要进行2次比较……因此，这个算法将进行“0+1+2+3+…+n”次比较。根据我们在第2章“程序设计面试题的解答思路”里的讨论 这个算法的执行时间将是$O(n^2)$级的。

这是你采用这一方案时所能得到的最佳结果。还有没有更好的方案呢？有，但在未经提示的情况下，人们是很难想到这个方案的，它需要用到两个指针。那么，有什么事情是使用一个指针无法完成，因而必须要用两个指针才能做到的呢？首先，你可以让它们交替前进，但这与使用一个指针的情况岂不是没有区别吗？其次，你可以让它们按固定间隔同时前进，可这也不会让你得到什么好处。那么，如果你让这两个指针以不同的步长前进会怎么样呢？

在非循环链表上，步长较大的指针（我们不妨称之为快指针）将率先到达链表尾。在循环链表上，这两个指针都将进入死循环，但快指针迟早会超过那个步长较小的指针（我们不妨称之为慢指针）。于是，如果快指针超过了慢指针，就说明那是一个循环链表；如果快指针遇到了“NULL”指针，就说明那是一个非循环链表。我们可以把它表述为如下所示的形式：

```
让快、慢两个指针从链表的头元素出发开始遍历
无限循环
    如果快指针遇到了“NULL”指针
```

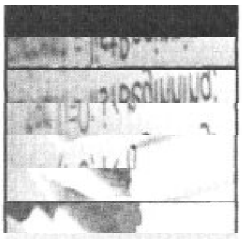
返回，该链表以“NULL”结束，是一个非循环链表
如果快指针追上或者超过了慢指针
返回，该链表是一个循环链表
让慢指针前进一个结点
让快指针前进两个结点

这个算法的代码实现如下所示，请注意“if (!fast || !fast->next)”语句中逻辑运算符“||”的用法。

```
/* Takes a pointer to the head of a linked list and determines if
 * the list ends in a cycle or is NULL terminated
 */
int DetermineTermination(node *head)
{
    node *fast, *slow;
    fast = slow = head;
    while (1) {
        if (!fast || !fast->next)[4]
            return 0;
        else if (fast == slow || fast->next == slow)
            return 1;
        else {
            slow = slow->next;
            fast = fast->next->next;
        }
    }
}
```

这个算法比我们讨论过的其他算法快吗？如果这是一个非循环链表，快指针将在遍历了$n$个结点后到达链表尾，此时慢指针只遍历了$n/2$个结点；因为总共遍历了（$n + n/2 = 1.5n$）个结点，所以这个算法将是O($n$)级的。

这个算法在循环链表上的执行时间又是怎样的呢？慢指针遍历链表尾部循环体的次数绝不会多于一次。当慢指针遍历到第$n$个结点的时候，快指针将遍历过$2n$个结点；所以不管链表尾部的循环体包含多少个结点，快指针肯定已经追上或者超过慢指针了。也就是说，你最多需要遍历$3n$个结点，因此该算法仍将是O($n$)级的。总之，就这道面试题而言，不论是循环链表还是非循环链表，采用两个指针的方案都要优于只采用一个指针的方案。



# 树和图

树（tree）和图（graph）都是程序设计工作中比较常见的数据结构，所以它们出现在程序设计面试的试题里也就不足为奇了。与树有关的面试题可能会更多见一些，因为它们能让面试考官快速了解你对递归技术和执行时间分析技术的掌握程度。此外，树相对要简单一些，在程序设计面试时间内，你怎么着也应该能弄出个结果了。而与图有关的问题虽然都很值得研究，但往往过于复杂，因而不适合充当程序设计面试中的试题。因此，这一章的讨论重点将主要以树为主。

## 4.1 树

树是由结点（或者叫数据元素）构成的，每个结点可以有零个、一个或者多个子指针指向其他的结点，但指向一个结点的指针最多只能有一个。

根据这一定义，我们将得到一个如图4-1所示的数据结构。

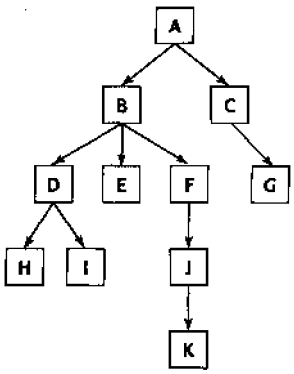



图4-1 树及其结点

类似于链表中的情况，树结点的指针和数据也都绑定在一个struct（C语言）、class（C++或Java）、或某种类似的结构里。下面是C语言对一个元素为整数的树结点的类型声明：

```
typedef struct nodeT {
    /* struct nodeT** because it points to an array of struct nodeT* */
    struct nodeT **children;
    int value;
} node;
```

在这个定义里，children指向一个指针数组，里面保存着指向该结点所有子结点的指针。

与树有关的问题可以用任何一种允许使用指针的程序设计语言来解决。但程序设计面试中最常用的还要说是C语言，所以我们将用C语言来解答本章中的面试例题。

请仔细观察一下图4-1中的树。树只有一个最高层次的结点；从这个结点出发，你可以沿着它的子指针到达任何一个其他的结点。我们把这惟一的最高层结点称为这个树的“根”（root）。只有从根结点出发，才可以保证你能到达其他任何一个结点。不管是什么样的树，都必须有一个根结点。因此，人们也经常用“树”这个词来指称树的根结点。

下面是一些与树有关的词汇：

**父结点**（parent）：如果某结点有一个指针指向另外一个结点，那么前者就是后者的父结点。除根结点外，每个结点都有一个父结点。在图4-1里，结点B就是结点D、E、F的父结点。

**子结点**（child）：如果某结点有一个指针指向另外一个结点，那么后者就是前者的子结点。在图4-1里，结点D、E、F都是结点B的子结点。

**后代**（descendant）：从某结点出发且沿着由它的子结点所构成的路径能够到达的全体结点叫做该结点的“后代”。在图4-1里，结点D、E、F、H、I、J、K都是结点B的后代。

**祖先**（ancestor）：从某结点出发，如果沿着由它的子结点构成的某条路径能够到达另外一个结点，那么前者就叫做后者的“祖先”。在图4-1里，结点A、B、D都是结点I的祖先。

**叶**（leaf）：没有任何子结点的结点。在图4-1里，结点G、H、I、K都是叶结点。

### 4.1.1 二元树

我们刚才给出的是关于树的最通用的定义。在实践中，当面试考官说“树”的时候，他通常指的是一种特殊类型的树，即所谓的“二元树”（binary tree）。二元树的每一个结点最多允许有两个子结点，人们在习惯上把它们称为左结点和右结点。图4-2给出了一个二元树的示意图。

参照前面对树结点做出的类型声明，我们可以得到关于二元树的结点定义，如下所示：

```
typedef struct nodeT {
    struct nodeT *left;
    struct nodeT *right;
    int value;
} node;
```

如果某结点没有左结点或右结点，相应的子指针就将为“NULL”。

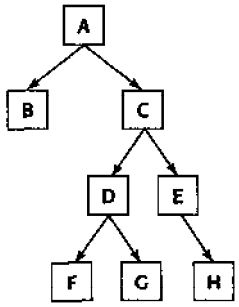


图4-2 二元树

与那些涉及普通树的同类问题相比，解答只涉及二元树的问题所需要花费的时间往往更短，但挑战性却丝毫不差。为了节约程序设计面试的宝贵时间，与树有关的面试题大都会以二元树的面目出现。因此，如果面试考官只是简单地说了一句“树”的话，你最好问清楚他指的是一个普通类型的树，还是一个二元树。

提示：在说到“树”的时候，人们的意思往往是指“二元树”。

### 4.1.2 二元搜索树

树经常被人们用来存放经过排序或者本身就是按一定顺序排列的数据。到目前为止，用树来保存数据的最常见的办法是使用一种特殊的树，即“二元搜索树”（binary search tree，简称BST）。在BST里，左结点的值永远小于或者等于其父结点的值，而右结点的值则永远大于或者等于其父结点的值。图4-3给出了一个二元搜索树的示意图。

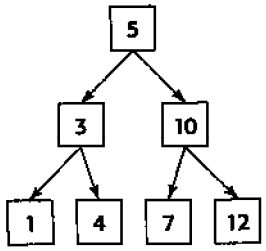


图4-3 二元搜索树

BST是最常用的一种树，所以在说到“树”的时候，人们的意思往往是指“二元搜索树”。

提示：在说到“树”的时候，人们的意思往往是指“二元搜索树”。

**查找**

二元搜索树的优势在于它的查找操作——在树里寻找某特定结点的操作——既快速又简单，非常适用于大量的数据存储。BST上的查找操作可以用自然语言描述为以下步骤：

```
从根结点出发
在当前结点不为“NULL”时循环
    如果当前结点的值等于你想查找的值
        返回当前结点
    如果当前结点的值小于你想查找的值
        前进到左结点
    如果当前结点的值大于你想查找的值
        前进到右结点
结束循环
```

如果直到循环结束也没有找到你想要的东西，就说明这个结点在这棵二元搜索树上不存在。

比如说，如果想把“6”找出来，就必须像下面这样做：

```
node *FindSixNode(node *root)
{
    node *curNode = root;
    while (curNode) {
        /* You've found the curNode */
        if (curNode->value == 6) return curNode;
        else if (curNode->value < 6) curNode = curNode->right;
        else if (curNode->value > 6) curNode = curNode->left;
    }
    return NULL;  /*No appropriate node exists */
}
```

这种以相应地前进到左结点或右结点的方式而进行的查找在速度上是非常快的，因为每次循都会筛掉一半的结点。在最坏的情况下，你将在只剩最后一个结点的时候知道自己的这次搜索是否成功。也就是说，查找操作的执行时间相当于你在到达最后一个结点之前所遍历过的结点的个数。假设这个数字是x，那它应该满足等式$2^x=n$。这是一个对数算式。例如，因为$2^3=8$，所以$\log_2 8=3$。因此，查找程序的执行时间将是$O(\log_2(n))$——人们习惯于省略对数中的底“$2^1$”而把它写成$O(\log(n))$的形式（不同底数的对数值呈固定的倍数关系，所以“大O记号法”省略了它们）。如果你知道$\log_2 1\,000\,000\,000 \approx 30$，就该明白$O(\log(n))$级的算法有多快了。

**提示：** 二元搜索树上的查找操作的执行时间是$O(\log(n))$级的。

不过，BST上的查寻操作并非总能达到$O(\log(n))$水平。只有在每次循环的确能筛掉一半或者接近一半的剩余结点时，它才是$O(\log(n))$级的。在最坏的情况下，BST的每个结点都将只有一个子结点，从而使BST实际成为了一个链表，而查找操作也将在这个“链表”上进行。我们知道，链表上的查找操作是$O(n)$级的。幸好人们早就注意到了这个问题，并且已经找到了一些能够让结点平均分布在BST左、右分支的办法（“红黑树”（red-balck tree）是这些办法中最常用的）。人们把左、右分支上的结点个数大致相等的树称为“平衡树”（balanced tree）。平衡BST上的删除和插入操作的执行时间也是$O(\log(n))$级的，具体分析过程我们就不在这里罗嗦了。

**提示：** 二元搜索树上的删除和插入操作的执行时间是$O(\log(n))$级的。

二元搜索树还有其他一些重要的特点。比如，1）如果一直沿着左指针前进，就能找到树上的最小值元素；如果一直沿着右指针前进，就能找到树上的最大值元

素；2）排序输出全部结点的操作执行时间是O($n$)；（3）任意给定一个结点，找出其父结点的执行时间将是O(log($n$))。

一般来说，与树有关的面试题的考察重点是求职者的递归思维能力。树上的每一个结点都是以该结点开始的那棵子树的根结点，这种从叶结点向根结点逐层嵌套的特点恰好与递归算法由下而上、由内而外、由小到大解决问题的特点相吻合。在与树有关的递归算法中，我们从根结点开始，执行一个动作，然后再前进到它的左子树或者右子树（或者依次前进到两棵子树）；这个过程将反复循环，直到遇上一个“NULL”指针——也就是到达树的根结点——为止。下面就是我们用递归算法实现的BST查找操作：

```
node *FindSixNode(node *root)
{
    if (!root) return NULL;
    else if (root->value == 6) return root;
    else if (root->value < 6) return LookupSix(root->right);
    else if (root->value > 6) return LookupSix(root->left);
}
```

与树有关的问题大都具备递归的特点。所以，到你遇到一个涉及到树的问题时，从递归的角度入手来对它进行分析通常都会有所收获。

**提示：** 与树有关的操作大都能用递归算法来实现。

### 4.1.3 堆

堆（heap）是另外一种比较常见的树。各种堆都是二元树（但堆的数据实现却可能与我们前面介绍的数据结构不一样）。堆的特点是各结点的值都小于它的父结点的值，所以堆的最大值出现在它的根结点上。堆的最大优点在于最大值查找操作的执行时间是一个常数——直接返回它的根结点就行了。堆的插入和删除操作的执行时间仍是O(log($n$))级的，但查找操作的执行时间却变成O($n$)级的了。与BST不同，堆无法在O(log($n$))时间内找到给定结点的父结点，也不能在O($n$)时间内排序输出全部的堆结点。

你可以用堆来为医院的候诊室建立一个模型。每进来一位病人，你就给他分配一个相应的优先级并插入到堆里去。一位心脏病患者的优先级当然要比一位只是崴了脚的病人的优先级来得高。当一位医生诊断完前一位病人后，候诊室里优先级最高的病人将最先被请入他的诊室。医生是靠什么来判断病人优先级的呢？很简单，他只要查一下候诊室堆模型里的最大值就行了——优先级最高的病人将出现在堆模型的根结点处，而堆的最大值查找时间是一个常数，这就不至于耽误了病人的病情。

**提示：** 在需要快速查出最大值的场合，请考虑使用堆这种数据结构。

### 4.1.4 常用的搜索方法

如果已知条件里给出的是一棵二元搜索树或堆，那你当然就用不着花时间去对它们进行排序了。但这种好事并不会天天发生，你拿到的可能是一棵代表家族谱系或者企业职务级别的树。要想从这类树里检索数据，就必须选用其他的办法。一种比较常见的问题是让你从中找出某个特定的结点。人们经常使用两种搜索方法来完成这项工作。

**宽度优先搜索**

对树进行搜索的办法之一是进行“宽度优先搜索”（breadth-first search，简称BFS）。在BFS算法里，你将从根结点出发，按从左到右的顺序先遍历第二层，再按从左到右的顺序遍历第三层……如此这般，直到你遍历完所有的结点或者找到你想要的结点为止。这个算法的执行时间为O($n$)级，所以最好不要用这个算法来搜索尺寸很大的树。BFS算法的内存开销也比较大，因为它在搜索每一层的同时需要把该层结点的子结点指针全都保存起来。

**深度优先搜索**

另一种结点搜索办法是进行“深度优先搜索”（depth-first search，简称DFS）。深度优先搜索将沿着树的某个分支一直向下搜索尽可能多的层，直到找到目标结点或到达这一分支的尽头；当搜索工作到达这一分支的尽头时，它将从距离最近且有子结点尚未被搜索过的祖先处开始继续搜索。与BFS相比，DFS的内存开销要小得多，因为不需要把各层结点的子指针统统保存起来。此外，DFS不会特意把某一层留到最后才去搜索（在BFS里，树的最低层将最后被搜索）；如果你怀疑自己正在寻找的结点位于树的最低层，那么使用DFS算法的搜索效果应该会更好一些。比如说，假设你想在一个企业职务级别树上查找一位参加工作还不到三个月的员工，就应该假设代表这位员工的结点是位于企业职务级别树中比较低的层次上。如果你的假设是对的，那么使用DFS通常要比使用BFS能够更快地找到目标结点。树的搜索方法还有很多，但我们在这里介绍的两种是求职者在程序设计面试中最有可能会遇上的。

### 4.1.5 遍历

与树有关的的另一类常见问题是对一棵未经排序的树进行遍历。与搜索操作在找到目标结点后就停止前进的做法不同，遍历操作必须完成对每一个结点的访问和处理。树的遍历方法也有很多，它们之间的差异无非是到达各个结点的顺序有所不同而已。三种最常用的遍历方法是左遍历（preorder）、中遍历（in-order）、右遍历（postorder）。虽然你对“遍历”这个词很熟悉，面试考官也很可能会让你写一段代码来实现对树的遍历操作，但你不太可能把各种遍历方法的细节都记忆无误。还好，面试考官通常都会在试题里把这方面的要求讲清楚。如果面试题的解决方案需要进

行遍历操作，你应该首先考虑能不能用递归的方式来完成之。

提示：如果面试考官要求你实现一种遍历方法，你应该首先考虑能不能用递归的方式来完成。

## 4.2 图

图比树要复杂得多。图上的结点也带有零个、一个或者多个指向其他结点的指针。图与树的不同之处在于1）往往会有多个指针指向同一个结点，甚至会形成一个循环圈；2）结点之间的连接路径往往带有一定的数值或者叫权（weight），这些路径的作用并不仅仅是充当指针，所以人们把它们称为“边”（edge）。图里的边可以是单向的，也可以是双向的。如果一个图里有单向的边，我们就称这个图为“单向图”（directed graph）。如果图里的边全都是双向的，我们就称之为“双向图”（undirected graph）；在双向图里，边两端的箭头一般用不着特意画出来。图4-4给出了一个单向图的例子，图4-5给出了一个双向图的例子。

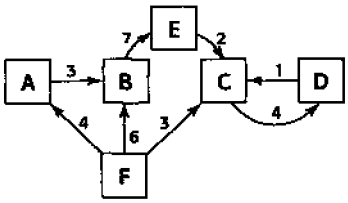



图4-4 单向图

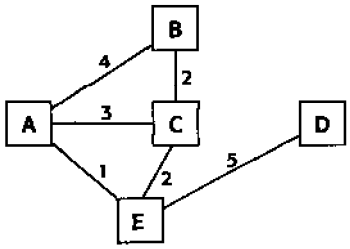



图4-5 双向图

现实世界里的事物通常都很复杂，有些事物很难用其他数据结构建立起数学模型来，用图则相对要简单得多。比如说，用单向图来建立一个城市供水管道系统的模型就很适合；你可以利用这个图来解决城市供水的最短路径问题。双向图也能为很复杂的事物——比如信号传输中继站等——构建模型。

与树相比，图这种数据结构在程序代码中的类型定义没有任何规律可循，而且往往由所选用的算法来决定。现实世界里的程序设计与图的关系相当密切，但对时间有限的程序设计面试来说，与图有关的问题实在是过于复杂了。因此，图很少会出现在程序设计面试中；对它有以上的了解应该足够了。

## 4.3 面试例题：左遍历

- **在这里，我们把树的左遍历定义为：从根结点出发，按逆时针顺序对全体结点进行遍历，每到达一个结点，就输出它的数值。比如说，对图4-6中的树进行**

**左遍历将依次输出 100、50、25、75、150、125、110、175。请实现一个对二元搜索树进行左遍历的算法，按遍历顺序输出各结点的数值。这个函数的调用接口必须是下面这样的：**

```
void PreorderTraversal(node *root);
```

想找到一个能以正确的顺序输出结点数值的算法，必须先把整个过程梳理清晰：先尽可能地向左前进，当无法继续前进时转到树的上一层，向右前进一个结点，再尽可能地向左前进，再转到上一层，直到遍历完全部结点为止。你可以从子树的角度来分析这道试题。

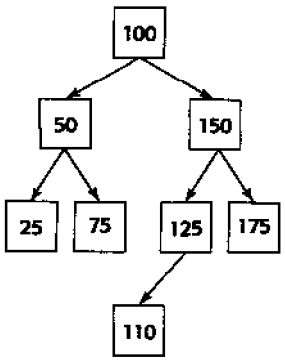


图4-6 二元搜索树

50和150分别是这棵树最大的两个子树的根。这两个子树上的结点有几个非常重要的特点：1）以50为根的子树上的结点将先于以150为根的子树上的结点被输出出来；2）根结点将先于子树上的任何结点被输出出来。归纳一下：当左遍历到某个结点时，你先输出该结点，然后左遍历到它的左子树，再左遍历它的右子树。如果从根结点开始说起的话，我们就能得到下面这样的递归性描述：

*输出根结点（或子树的根结点）的值*
*对左子树进行左遍历*
*对右子树进行左遍历*

把这个描述翻译成C语言代码，我们就得到：

```
void PreorderTraversal(node *root)
{
    if (root)
        printf("%d\n", root->value);
    else
        return;
    PreorderTraversal(root->left);
    PreorderTraversal(root->right);
}
```

这个算法的执行时间是多少呢？因为每个结点都被检查了一次，所以它是O(*n*)级的。

## 4.4 面试例题：左遍历，不使用递归

- **对二元搜索树进行左遍历并输出各结点的值。函数的调用接口不变，但不允许使用递归方法。**

```
void PreorderTraversal(node *root);
```

不使用递归，这道试题该怎样来解决呢？有些任务可以用递归算法来解决，也可以用非递归算法来解决，但两种算法在工作原理和数据结构方面往往会有很大差异。把自己知道的数据结构好好想一遍，看哪一种最适合这个场合。你可以拿列表（list）、数组或其他二元树试试。不过，左遍历本身就是一种递归性的操作，所以递归性的左遍历算法是很难用一个完全不同的循环性算法来替代的。因此，我们认为深入分析一下递归算法的本质，再用循环算法来模拟它的动作将是此时的最佳策略。递归算法用到了堆栈这种数据结构，利用堆栈来完成数据处理。如果我们能“制造”出一个堆栈——先别管堆栈是怎样实现的——来存放二元搜索树的结点指针，事情就好办多了。堆栈应该具备以下四个函数（如果不清楚这几个函数的用途，请重温第3章中的面试例题“堆栈的实现”）：

```
int Push(element **stack, void *data);
int Pop(element **stack, void **data);
int CreateStack(element **stack);
int DeleteStack(element **stack);
```

我们再仔细研究一下递归算法的流程。只要把隐含在递归算法里的堆栈用法弄清楚了，我们就能明确地用循环算法来模拟出用堆栈来保存数据的效果来。

递归算法是这样的：

*输出根结点（或子树的根结点）的值*

*对左子树进行左遍历*

*对右子树进行左遍历*

第一次进入递归函数的时候，你先输出根结点的值，然后递归地调用这个函数对左子树进行遍历。在发出这个递归调用的时候，调用者的程序状态就会被压入堆栈保存起来，这样，在递归调用返回时，调用者就能从自己刚才离开的地方开始继续执行——就左遍历算法来说，调用者将继续去对右子树进行左遍历。为了能在遍历完左子树之后开始对右子树的遍历，递归调用隐含地把右子树的地址保存到了堆栈上。在输出一个结点之后但在前进到它的左结点之前，该结点的右结点将被压入一个你不知道的堆栈里保存起来。当没有子结点可供处理时，你就会从递归调用中返回一层，从堆栈上弹出一个右结点，然后以它为起点继续进行左遍历。总结一下：这个算法先输出当前结点的值，再把它的右结点压入堆栈，然后前进到它的左结点；当没有子结点可供处理（即到达一个叶结点）时，算法将从堆栈上弹出一个新的当前结点。这一过程将一直重复到遍历完所有结点为止，而堆栈里也将不再有任何数据。

在动手实现这个算法之前，应该先把容易出乱子的各种特例考虑周全。你不必分别编写两个例程来分别压入指向左、右两个结点的指针，最好是用一个例程来完成这两个动作。请注意左、右结点被压入堆栈的顺序——必须保证左结点总是先于右

结点被弹出来。

堆栈是一种后进先出的数据结构，所以你可以先压入右结点，再压入左结点。这样，你就用不着再明确地前进到左结点去了——你可以先弹出堆栈上的第一个结点，输出它的值，再按先右后左的顺序把它的两个子结点压入堆栈。如果这一系列动作是从二元搜索树的根结点开始——即先压入根结点，再按刚才描述的顺序来弹出当前结点、输出数值、压入子结点——的话，你就能精确地模拟出左遍历递归算法的过程。在添加一些细节后，我们得到了下面这个算法：

```
创建堆栈
把根结点压入堆栈
当堆栈不为空时，循环
   弹出一个结点
   如果这个结点不是“NULL”
      输出它的值
      把这个结点的右结点压入堆栈
      把这个结点的右结点压入堆栈
```

下面是这个算法的实现代码（省略了诸如堆栈无法分配到内存之类的出错情况检查）：

```
void PreorderTraversal(node *root)
{
    element *theStack;
    void *data;
    node  *curNode;

    CreateStack(&theStack);
    Push(&theStack, root);

    while (Pop(&theStack, &data)) {
        curNode = (node *) data;
        if (curNode) {
            printf("%d\n", curNode->value);
            Push(&theStack, curNode->right);
            Push(&theStack, curNode->left);
        }
    }
    DeleteStack(&theStack);
}
```

这个算法的执行时间是多少呢？每个结点只被处理了一次、被压入了堆栈一次，所以它是一个O($n$)算法。这个算法的优点是没有函数调用方面的开销，缺点是因模拟堆栈操作而增加了动态内存分配方面的开销。递归算法与循环算法的执行效率到

底谁高还不能下结论。

## 4.5 面试例题：最低公共祖先

- **已知二元搜索树上两个结点的值，请找出它们的最低公共祖先。你可以假设这两个值肯定存在。这个函数的调用接口如下所示：**

```
int FindLowestCommonAncestor(node *root, int value1,
                             int value2);
```

**请看图4-7里的二元搜索树，如果4和14是给定的value1和value2，它们的最低公共祖先就将是8——它既是4的祖先，也是14的祖先，并且再也没有比8更低层的结点是4和14的共同祖先了。**

根据图4-7，我们可以立刻得到一个这样的算法：从两个给定的结点出发进行回溯，两条回溯路线的交点就是我们要找的东西。这个算法的具体实现办法是：先用这两个结点的全体祖先分别生成两个链表，再把这两个链表第一次出现不同结点的位置找出来，则它们的前一个结点就是我们要找的东西。这个解决方案本身倒没有什么不好的地方，只是我们还能找出一个更具执行效率的算法来。

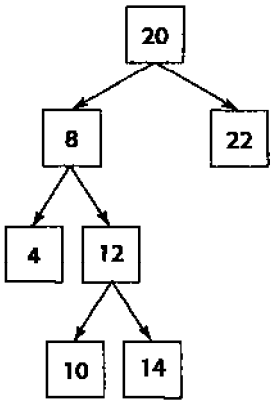


图4-7 最低公共祖先

这第一个算法没有利用到二元搜索树的任何特性，其他类型的树也可以用这个算法来处理。你应该研究一下二元搜索树的特点，找出一个更有效的最低公共祖先寻找算法。

二元搜索树有两个特点。其一，每个结点有零个、一个、或两个子结点；但我们似乎无法利用这一特点来得到一个更有效的算法。其二，左结点的值永远小于或等于当前结点的值，而右结点的值则永远大于或等于当前结点的值。这个特点好像有点利用价值。

我们再仔细研究一下图4-7。4和14的最低公共祖先是8，它与4和14的其他公共祖先是有重要（但不容易看出来）区别的：其他的公共祖先或者同时大于4和14，或者同时小于4和14；只有8是介于4和14之间的。利用这一研究成果，我们就能得到一个更好的算法。

根结点是一切结点的祖先，从根结点出发可以到达任何一个结点。因此，我们将从根结点出发，沿着两个给定结点的公共祖先前进。当这两个结点的值同时小于当前结点的的值时，沿左指针前进；当这两个结点的值同时大于当前结点的的值时，沿右指针前进；当第一次遇到当前结点的值介于两个给定的结点值之间的情况时，

这个当前结点就是我们要找的最低公共祖先了。

根据以上描述，我们得到一个下面这样的算法：

检查当前结点

如果value1和value2同时小于当前结点的值

前进到当前结点的左结点

如果value1和value2同时大于当前结点的值

前进到当前结点的右结点

否则

当前结点就是我们要找的最低公共祖先

这是一道与树有关的试题，算法也有递归的味道，所以用递归来实现这一解决方案似乎是顺理成章的事。可这里并没有这个必要。递归技术特别适用于对树的多个层次进行遍历或需要寻找某些个特殊结点的场合。这道面试题只是沿树结点逐层向下前进，用循环语句来实现有关过程将更简单明了。如下所示：

```
int FindLowestCommonAncestor(node *root, int value1,
                             int value2)
{
    node *curNode  = root;

    while (1) {
        /* Go to the left child */
        if (curNode->value > value1 && curNode->value > value2)
            curNode = curNode->left;

        /* Go to the right child */
        else if (curNode->value < value1 &&
                 curNode->value < value2)
            curNode = curNode->right;

        /* Else, you've found the correct node */
        else
            return curNode->value;
     }
}
```

这个算法的执行时间是多少呢？你直接沿着由两个给定结点的公共祖先构成的路线前进，而沿任一路线到达树的某一结点的算法是O(log($n$))级的。因此，这个算法也将是一个O(log($n$))级算法。此外，与相应的递归算法相比，本算法的执行效率要更高一些，因为它没有反复进行函数调用的开销。

# 数组与字符串

数组和字符串是计算机数据世界里的“中层干部”，它们比整数、浮点数等简单的单数值类型来得复杂，比类（class）或各种动态数据结构来得简单。各种程序设计语言中的数组有一些共同的特点，但在具体实现方面又各有特色。与数组相比，字符串对具体程序设计语言的依赖程度更大。字符串与数组之间有着非常密切的关系，特别是在C和C++当中。

## 5.1 数组

数组是由一组同类型的变量构成的，它们将被存放在一块连续分布的内存区域里。数组在各种主流的程序设计语言里都扮演着一个十分重要的角色。做为参加程序设计面试的求职者，相信大家对数组的语法和用途应该已经有很深的体会了，所以我们对数组的讨论将主要集中在与字符串有关的理论和应用方面，这也是数组的重要应用领域之一。

像链表一样，数组也是一种线性的存储结构（多维数组是一种多维的线性结构），但这两种数据结构有着本质的区别。在链表中查找元素是一个O($n$)操作；在数组中查找元素则是一个O(1)操作——如果你知道那个元素的下标的话。下标的意义十分重大，如果你只知道元素的值，那么查找操作仍将是一个O($n$)操作。比如说，给定一个字符数组，查找它第6个元素的操作是O(1)操作，而查找取值为 'w' 的字符则是一个O($n$)操作。

数组的查找操作的效率提高是以牺牲数组的数据插入和删除操作的效率为代价的。因为数组元素都存储在一段连续的物理内存空间里，所以在插入或者删除它的某个元素时，就不能像对待链表那样简单地分配或者释放该元素所占据的空间；你必须用挪动数组中其他数据的办法来为它腾出一个地方或者填补它留下的空缺。

数组一种非动态的数据结构，它里面的元素个数是一个有限的固定数字。即使你只用到数组的一小部分，也必须为全体数组元素分配空间。数组特别适用于你在程序开始运行之前就已经知道需要用到多少个元素的场合。如果程序用到的存储空间量是变化的，使用数组就不方便了，因为存放到数组里的数据量将不得超过它的容量。当然，你可以用一个大数组来确保程序用到的数据量低于它的容量，但这种做法往往会浪费大量的内存。

为了解决这一问题，人们提出了动态数组（dynamic array）的概念。动态数组是一种动态的数据结构，它既具备数组的特点，又能根据数据的实际用量来调整自己的尺寸。我们不打算深入探讨动态数组的实现细节，只想告诉大家一个重要的事实：动态数组大都是以静态数组做为其内部实现手段的。静态数组的尺寸当然不允许改变，所以调整动态数组尺寸的工作其实是这样进行的：先分配一个尺寸适当的新数组，再把老数组里的元素拷贝到新数组里，然后再释放老数组所占用的空间。这是一项开销巨大的操作，所以如果你的程序用到了动态数组，就应该尽可能地少去调整它的尺寸。

不同的程序设计语言在对数组的处理上有着细微的区别，有些方面比较强，有些方面就比较弱。下面，我们将对C/C++、Java、Perl三种语言中的数组进行一下对比，看它们各自在哪些方面容易出问题。

### 5.1.1 C / C++

撇开C与C++之间的区别，它们在对数组的处理方面是很相似的。在大多数场合里，一个数组名就相当于一个指向该数组第一个元素的指针常数。（指针和常数都是比较容易弄混的概念，把它俩放到一起就更不好掌握了。在说到“指针常数”时，我们的意思是一个被声明为“char *const chrPtr”形式的指针，你不能把它修改为指向内存中的另一处地址，但可以用它来改变它所指向的地址处的数据内容。这与更为常见的“常数指针”是有区别的。在说到“常数指针”时，我们的意思是一个被声明为“const char *chrPtr”形式的指针，你可以把它修改为指向内存中的另一处地址，但不能用来修改它所指向的地址处的数据内容。如果你觉得有点糊涂，请不要慌张；因为你肯定不是这些人中的第一个，也不会是最后一个。《Expert C Programming》一书对这类问题做了细致深入的分析和讲解；这本书的作者是Peter Van Der Linden，由Prentice Hall出版公司1994年出版。）也就是说，你不能通过一个简单赋值语句用一个数组来初始化另一个数组中的元素。

比如说，当编译器遇到下面这条语句时：

```
arrayA = arrayB;  /* Compile error: arrayA is not an lvalue */
```

它会认为你想让arrayA和arrayB指向同一块内存区域。如果你已经把arrayA声明

为一个数组，就会引起一个编译错误，因为C语言不允许你把arrayA修改为指向另一块内存区域。如果你真的想把arrayB复制到arrayA里去，就必须用一条循环语句来一个一个地复制有关元素，或者用memcpy之类的库函数来完成这一工作。

在C和C++里，编译器只检查数组的起点位置，不检查它的尺寸边界；是否超出数组边界的检查全部要由程序员来负责。在存取数组元素的时候，编译器不检查它是否已经超出了该数组的边界。这么说吧，即使某数组只有10个元素而你却在对它的第20个元素进行存取，C语言也不会提示出错。你的操作将“侵犯”其他的数据结构并引发各种各样的问题，这类bug往往很难被发现和纠正。为了帮助程序员查找C语言程序中的数组访问越界等与内存有关的问题，人们已经开发出了很多种开发辅助工具。

### 5.1.2 Java

与C程序中的数组不同，Java程序中的数组是一种特殊的指针类型，不能与指向数组元素的指针互换使用。与C语言中的情况一样，Java数组也不能用一条简单赋值语句来拷贝。如果两个数组指针的类型相同，把它们中的一个赋值给另一个是允许的，但效果却是使两个指针都指向同一个数组，如下所示：

```
byte arrayA[] = new byte[10], arrayB = new byte[10];
arrayA = arrayB; // arrayA now refers to the same array as arrayB
```

Java内建有对数组的尺寸边界进行检查的功能，如果你试图访问的数组元素超出了该数组的边界，Java就将抛出一个“ArrayIndexOutOfBoundException”（数组下标越界例外）。

Java数组最古怪的特点是：在给一个对象数组分配内存时，它并不会把有关的对象构造出来；操作员必须亲自去构造这些对象并依次赋值给该数组中的各个元素，如下所示：

```
Button myButtons[] = new Button[5]; // Buttons not yet constructed
for (int i = 0; i < 5; i++) {
    myButtons[i] = new Button();  // Constructing Buttons
}
// All Buttons constructed
```

### 5.1.3 Perl

Perl数组是一种功能非常强的数据结构。对Perl数组的全面介绍超出了本书的范围，这里只把Perl数组与其他程序设计语言中的数组之间的主要区别重点说明一下。

Perl的独特之处在于它根本没有静态数组——它的array类型就是一个动态数组。当数据访问操作超出数组边界时，它将简单地调整该数组的尺寸并让这个操作落在数组的边界以内。同时，因为枚举（scalar）是Perl语言中惟一的简单数据类型，所

以它的数组类型也只有一种。这种安排有别于其他大多数程序设计语言，在其他的程序设计语言里，一个浮点数组与一个整数数组往往是不同的类型。

与那些从C语言发展出来程序设计语言不同，Perl允许你通过一条简单赋值语句用一个数组来初始化另一个数组，它们仍将是两个数组，只是内容相同而已，如下所示：

```
@arrayA = @arrayB; # still two separate arrays, now with same contents
```

在与Perl数组打交道的时候，让数组的名字错误地出现在枚举性上下文中要算是最容易犯的错误了。此时，数组名的值将是它的尺寸长度，而不是它元素的值。如下所示：

```
@a = (1, 2, 3);
print @a;  # prints 123
print @a + 2; # + creates scalar context, prints 5 instead of 345
```

## 5.2 字符串

字符串是由字符构成的一个序列。虽然把数组和字符串做为两种事物来对待，但大部分程序设计语言都把字符串保存为数组的形式。下面，我们将对C、C++、Java、Perl中的字符串进行一下对比。

### 5.2.1 C

C语言中的字符串其实就是一个char类型的数组。大家知道，C语言不检查数组的边界，所以C语言也不检查字符串的边界；但它会在字符串的末尾加上一个“NUL”字符（即字符'\0'）。字符数组必须为这个“NUL”字符留出一个位置。比如说，一个由10个字符组成的字符串需要一个至少11个字符长的数组来保存。这一机制使求取字符串长度的操作成为了一个O(*n*)级、而不是O(1)级的操作：用来求取字符串长度的库函数strlen()必须扫描到字符串的末尾才能计算出它的长度。

C语言不允许把一个数组赋值给另一个数组，同样的道理，C语言中的字符串也不能用赋值操作符“=”来拷贝；你得用库函数strcpy()来完成这一操作。

既然C语言里的字符串相当于数组，我们就能直接寻址到它里面的每一个字符，这大大方便了单个字符的读、改操作。如果你以这种方式改变了字符串的长度，有两件事请千万不要忘记：1）在新字符串的末尾加上一个“NUL”（即'0'）字符；2）确保字符串有足够的长度来容纳这个“NUL”字符。在C语言里，截短字符串的操作是很容易完成的——只要把“NUL”字符放到新字符串的末尾就行了。

### 5.2.2 C++

C++也像C语言一样以字符“NUL”做为字符串的结束符。C++是一种面向对象

的程序设计语言，用它写出来的字符串类（class）能够提供更丰富的功能，比如更简单的字符串合并操作、动态改变字符串的长度等。C++标准函数库里有一个名为“basic_string”的模板类（template class），为了增强字符串处理功能，C++开发工具大都在这个类的基础上进行了扩展，大家熟悉的“String”类就是其中之一。

### 5.2.3 Java

Java字符串是“java.lang.String”类的对象。String与字符数组可以相互转换，但却是不同的类型；可因为String类的方法（method）允许你读取字符串里的单个字符，所以读操作在这两种类型上的差异更主要地体现在语法而非功能方面。

两种类型在字符串写操作方面的差异可就比较大了。C字符串可以随机修改（即写操作），而Java程序中的String对象却是“百毒不侵”——被构造出来之后，其内容就不能改变。如果你想对一个String对象的内容进行操作，就必须从String构造出一个StringBuffer来。不过，这条限制并不像听上去那么苛刻。比如说，字符串合并操作符（+）能根据被合并字符串的内容自动构造出一个新的字符串来。StringBuffer只有当你需要一个字符一个字符地改变字符串的内容时才是必不可少的。

### 5.2.4 Perl

C语言中的字符串与数组几乎没有差别，Perl却走向了另一个极端——它把字符串当做一种枚举类型的数据。虽说split()函数能够轻而易举地把一个Perl字符串转换为一个字符数组，但很少有这种必要——因为Perl语言提供的字符串操作符和字符串函数非常充足。事实上，有很多人正是为了Perl强大的字符串和文本处理功能才来学习和使用这种语言的。其中最值得一提的是一套完备的基于规则表达式（regular expression）的搜索/替换操作符。比如说，如果我们想在自己的名字前面加上一个尊称，用一个简单的Perl操作符就能做到这一点，如下所示：

```
$Complaint = "On occasion, John and Noah rant incoherently.";
$Complaint =~ s/John and Noah/Mr. Mongan and Mr. Suojanen/;
print $Complaint;
# Prints:
# On occasion, Mr. Mongan and Mr. Suojanen rant incoherently.
```

## 5.3 面试例题：第一个无重复字符

- **请编写一个高效率的函数来找出字符串中的第一个无重复字符。例如，“total”中的第一个无重复字符是“o”；“teeter”中的第一个无重复字符则是“r”。请对你算法的执行效率做出评估。**

这道试题乍看上去好像没什么难度。重复字符至少会在字符串里出现两次，只要把字符串中的每个字符与其他字符比较一下，就能知道它是不是一个重复字符了。这一方案是很容易实现的：从字符串的第一个元素开始依次进行这种比较，当遍历到一个没有重复出现过的字符时，这个字符就是我们要找的第一个无重复字符。

这个算法的时间效率是怎样的？如果字符串有$n$个字符，那么，在最坏的情况下，这$n$个字符每一个都要做差不多$n$次比较，所以这个算法在最坏情况下是执行时间是$O(n^2)$。如果这个字符串只是一个单词，那你遇到最坏情况的可能性还不大；如果这个字符串是一段很长的文本，大多数字符就都可能会反复出现，遭遇最坏情况的概率将大大增加。虽然没有错误，可这个算法太容易想到了，你应该立刻想到它可能还有一个更好的解决方案——程序设计面试题的“标准”答案不应该这么简单浅显。肯定有一个算法的执行时间即使在最坏的情况下也要优于$O(n^2)$。（如果只把每个字符与它后面的字符相比较——因为它前面的字符都已经与它进行过比较了，这个算法的效率将得到一定的改善。总的比较次数将是“$(n-1)+(n-2)+\cdots+1$”次。不过，根据我们在第2章中的讨论，它仍是一个$O(n^2)$级算法。）

前面这个算法为什么是$O(n^2)$级的呢？这个$n^2$的因子之一来自该算法必须对字符串中的每一个字符进行检查以判断它是否重复出现。因为非重复字符可能出现在字符串中的任何一个位置，所以从这里入手来提高算法的效率似乎没有什么可能性。这个$n^2$的第二个因子来自该算法必须扫描整个字符串以判断某一个字符是否重复出现。如果能改善这个扫描操作的效率，整个算法的效率必将得到提高。要想改善一组数据上的扫描操作的效率，最简单的办法就是把这些数据放到一个允许进行高效率扫描的数据结构里去。哪种数据结构的扫描效率能够比$O(n)$更高呢？二元搜索树的扫描操作是$O(\log(n))$级的；数组和哈希表的元素查找操作都是固定时间。我们决定从数组或哈希表着手，因为这两种数据结构所提供的潜在改善是最大的。

我们需要快速判断某个字符是否在字符串中重复出现，所以需要用字符来搜索那个数据结构。这意味着我们需要使用字符做为数组的下标（C语言允许你把字符映射为整数去充当数组的下标）或者做为哈希表的键字。将被存放到这些数据结构里的数据又是什么呢？非重复字符只在字符串里出现一次，如果我们把字符在字符串中的出现次数记录下来，就能帮助我们找出字符串中的非重复字符。要想把每一个字符的出现次数统计出来，就必须对整个字符串进行扫描。

在完成这一步骤之后，我们将扫描数组或哈希表中的统计数字，把里面的“1”找出来。每个“1”都对应着一个非重复字符，但它未必是原字符串中的第一个非重复字符。

因此，我们需要按照各字符在原字符串里的出现顺序对其出现次数进行查找。这并不困难——只需依次查看每一个字符的出现次数直到遇上一个“1”为止；与我

们遇到的这个“1”相对应的那个字符就是所谓的第一个非重复字符。

我们来看看这个新算法的执行效率是否真的得到了提高。你必须遍历整个字符串才能建立起统计字符串出现次数的数据结构。在最坏的情况下，你可能要直到查遍字符串的每一个字符才能找到第一个非重复字符。因为数组或哈希表的查寻操作是固定时间，所以最坏情况将是两个步骤都要遍历字符串中的全部字符，即进行2n次操作，所以算法的执行时间将是O(*n*)级的——与刚才的算法比，这已经是很了不起的改善了。

哈希表和数组的元素查寻时间都是一个固定值，你需要在它们中间挑出一个来使用。哈希表的查寻开销略大于数组，但数组元素的初始化值是随机的——你必须先花时间把它们全部初始化为零才行。它们之间的最大区别也许是在内存的使用量方面了。采用数组的方案需要为每一种可能出现的字符准备一个元素以存放该字符出现次数的统计数字：如果你处理的是ASCII字符串，则数组中的元素个数将是256个；如果你处理的是Unicode字符串，数组中的元素个数将超过65 000个（因为Unicode是16位的字符）。再看哈希表，输入字符串中存在有多少种字符，哈希表中的元素个数就将是多少。所以，如果你处理的字符串很长，但它用到的字符集却很有限的话，数组就将是一个最佳选择；如果你处理的字符串比较短，但在其中出现过的字符种类却比较多的话，哈希表将是一个更优的选择。

两种解决方案都不错，你选择实现哪一种都行。我们的选择是哈希表方案——假设代码需要处理Unicode字符串。因为Java有内建的哈希表和Unicode支持功能，所以用Java来编写这个函数的代码要更容易一些。下面是关于这个算法的自然语言描述：

```
第一步，建立字符出现次数统计哈希表：
    针对字符串中的每一个字符
        如果哈希表中没有与该字符对应的统计数字，存入“1”
        否则，递加与它对应的统计数字
第二步，对字符串进行扫描：
    针对字符串中的每一个字符
        如果哈希表中的统计数字为“1”，返回该字符
    如果没有统计数字为“1”的字符，返回“null”
```

现在来编写函数代码。因为不知道这个函数将归属于哪一个类（class），所以我们将把它实现为一个“public static”（静态公共）函数——它等价于普通的C语言函数。你还必须注意，Java的Hashtable只能用来保存数据对象，所以保存在哈希表里的数据必须被声明为引用类型“Integer”而非基本数据类型“int”。代码如下所示：

```
public static Character FirstNonRepeated(String str)
{
    Hashtable charHash = new Hashtable();
```

```
    int i, length;
    Character c;
    Integer intgr;

    length = str.length();

    // Scan str, building hashtable
    for (i = 0; i < length; i++) {
        c = new Character(str.charAt(i));
        intgr = (Integer) charHash.get(c);
        if (intgr == null) {
            charHash.put(c, new Integer(1));
        } else {
            // Increment count corresponding to c
            charHash.put(c, new Integer(intgr.intValue() + 1));
        }
    }

    // Search hashtable in order of str
    for (i = 0; i < length; i++) {
        c = new Character(str.charAt(i));
        if (((Integer)charHash.get(c)).intValue() == 1)
            return c;
    }
    return null;
}
```

## 5.4 面试例题：删除特定字符

- **用C语言编写一个高效率的函数来删除字符串里的给定字符。这个函数的调用模型如下所示：**

```
void RemoveChars (char str[], char remove[] );
```

**注意，remove中的所有字符都必须从str中删除干净。比如说，如果str是“Battle of the Vowels: Hawaii vs. Grozny”，remove是“aeiou”，这个函数将把str转换为“Bttl f th Vwls: Hw vs. Grzny”。请对你的设计思路做出解释，并对你解决方案的执行效率进行评估。**

我们可以把这道试题分解为两个子任务。首先，对字符串str中的每一个字符，你必须判断出它是否应该被删除；然后，完成相应的删除操作。我们先来看看第二个任务——删除字符。

C字符串是保存在数组里的，所以这个任务就等价于删除数组中的某个元素。数组是一段连续的内存块，你不能像对待链表那样直接在数组的中间位置删除一个元素。你必须挪动数组中的其他元素以填补那个字符被删除后造成的空缺，使数组仍

然能保持为一个连续的内存块。比如说，如果删除了字符串“abcd”中的字符“c”，那么，你或者需要把字符“a”和“b”朝字符串尾的方向（为叙述方便，我们将称之为向“右”）挪一个位置，或者需要把字符“d”朝字符串头的方向（为叙述方便，我们将称之为向“左”）挪一个位置；两种办法都能使字符“abd”成为数组中连续排列的元素。除挪动字符外，你还需要把字符串的长度减少一个字符。如果是把字符朝右挪，就需要删除原来的第一个字符；如果是把字符朝左挪，就需要删除原来的最后一个字符。在C语言里，删除最后一个字符的操作是比较容易实现的——只要在新字符串的末尾写上一个NUL（即'\0'）字符就行了。在另一方面，删除字符串的第一个字符却没有这么容易。可见，从字符串删除某个字符之后，把剩余字符朝右挪将是最简明的选择。

这个算法在遇到最坏情况——即当你必须删除字符串str中的所有字符——时的表现如何呢？每删除一个字符，你都必须把剩余的字符全部挪动一遍。如果字符串str的长度是$n$个字符，则删除第一个字符时你需要挪动$n-1$个字符，删除第二个字符时需要挪动$n-2$个字符……所以删除操作的执行时间在最坏情况下将是$O(n^2)$。（如果你从字符串的尾部开始向前做删除操作，这个算法的效率会稍微高一些，但在最坏情况下仍将是$O(n^2)$。）因为要对同样的字符串做多次的挪动，所以这个算法的效率是相当低下的。怎样才能避开这一情况呢？

建立一个字符串临时缓冲区，把修改后的字符串拷贝到这个临时缓冲区，有这个办法来避免挪动那么多的字符；你觉得这个办法怎么样？你只需简单地把需要保留下来的字符拷贝到这个临时缓冲区里，把你准备删除的字符全都隔过去；然后，当你完成了对原字符串的处理并在临时缓冲区里得到最终结果时，再把它拷贝回字符串str里。这样，每个字符最多被挪动两次，这将使删除操作的执行时间变成$O(n)$。但临时缓冲区的长度与原字符串的长度是相等的，这增加了内存方面的开销；同时，字符和字符串的拷贝操作也增加了时间方面的开销。有没有办法让这个$O(n)$算法躲开这些开销呢？

在实现这个$O(n)$算法的时候，你必须用到一个指示着字符串str的读位置的源指针和一个指示着临时缓冲区中写位置的目标指针。这两个位置都是从零开始计算的。每在字符串str中读取一个字符，源指针就将前进一个位置；而每在临时缓冲区中写入一个字符，目标指针也将前进一个位置。换句话说，当你拷贝一个字符的时候，源指针和目标指针都将前进一个位置；而当你删除一个字符的时候，只有源指针会前进一个位置。也就是说，源指针或者与目标指针齐头并进，或者超前于目标指针。在你读取了源字符串中的某个字符之后（即源指针路过该元素并继续前进之后），你就用不着再保留这个字符了——事实上，你将只在最后需要把结果字符串拷贝回这个位置时再用它一次而已。因为源字符串中的目标位置永远是一个你已经不再会用到

的字符，所以你完全可以把目标字符直接写到源字符串里去——这将彻底省掉临时缓冲区。这仍是一个O(*n*)算法，但要比前一个版本的开销小多了。

把删除字符的事情研究明白之后，我们再来看看如何判断字符是否需要被删除。最简单的办法是把字符与字符集remove中的每一个字符进行比较，看它们是否匹配。这样做的效率是怎样的？如果字符串str的长度是*n*个字符，remove的长度是*m*个字符，那么，在最坏的情况下，你必须对*n*个字符中的每一个进行*m*次比较，所以算法的执行时间将是O(*nm*)。你必须对字符串str中的*n*个字符进行检查，那你能不能想个办法，让判断给定字符是否在字符集remove里的匹配操作优于O(*m*)呢？

如果你仔细研究过“面试例题1”的解决方案，就应该发现它与眼下这一局面有相似之处。类似于我们在“面试例题1”里的做法，你可以利用remove建立一个数组或哈希表，它们的查寻操作都只花费固定时间，进而引出一个O(*n*)解决方案。（你也许会问，为什么要建立一个数组呢？remove本身不就是一个数组吗？是的，它是一个数组，但它的下标只与字符位置有关，对解答这道试题毫无帮助；你仍需要遍历remove中的每一个字符才能断定str中的某个字符是否需要被删除。而我们所说的数组是一个以有关字符为下标的布尔数组；你可以根据这个数组的下标直接断定某字符是否在remove里。）在分析“面试例题1”的时候，我们已经对哈希表和数组的优、缺点进行了讨论。就眼下的这道试题来说，如果str和remove都比较长，但涉及的字符集却不太大（例如，两者都是ASCII字符串）的时候，数组将是最恰当的选择。如果str和remove都比较短、但涉及的字符集却很大（例如，两者都是Unicode字符串）的时候，哈希表将是更佳的选择。这一次，我们将假设我们处理的是ASCII字符串，所以选择了数组来代替哈希表。

用C语言写出来的函数分为三个部分。首先，把速查表数组里的全体元素设置为“false”。然后，遍历remove中的每一个字符，把它们在速查表数组里的对应元素设置为“true”。最后，用两个指针（一个源指针和一个目标指针）来遍历字符串str，对速查表数组里的对应元素为“false”的字符进行拷贝就能得到最终的结果。

现在，你可以把两个子任务合并到一个算法里去了。假设str的长度是*n*个字符，remove的长度是*m*个字符，下面对这个算法的执行效率进行一下分析。在任一给定的平台上，字符集的大小是一个固定的数字，把速查表数组中的全部元素初始化为零的时间也将是一个固定值。根据remove中每一个字符对速查表数组中的有关元素进行赋值的时间是一定的，所以建立速查表数组的时间是O(*m*)。最后，对字符串str中的每一个字符进行扫描和拷贝的操作时间也是一定的，所以这一阶段的执行时间将是O(*n*)。两部分加在一起将得到O(*n*+*m*)，所以我们这个算法的总执行时间是线性的。

在对算法进行完分析和评估之后，你就可以动手编写它的代码了。如下所示：

```
void RemoveChars(char str[], char remove[])
```

```
{
    int src, dst, removeArray[256];

    /* Zero all elements in array */
    for (src = 0; src < 256; src++) {
        removeArray[src] = 0;
    }

    /* Set true for chars to be removed */
    src = 0;
    while (remove[src]) {
        removeArray[remove[src]] = 1;
        src++;
    }

    /* Copy char unless it must be removed */
    src = dst = 0;
    do {      /* do..while terminates after copying NUL */
        if (!removeArray[str[src]]) {
            str[dst++] = str[src];
        }
    } while (str[src++]);
}
```

做为对照，下面是完成同一任务的Perl代码，它只有一条语句：

```
$str =~ s/[$remove]//g;[8]
```

（如果你是一位Perl语言的初学者，可能会疑惑这里为什么没有使用“tr///”操作符。我们选择“s///”操作符的原因是：“tr///”会在编译时建立一个转换表，这将不可避免地引起“eval”求值操作，从而增加整个操作的复杂性。当然，即便是使用了“tr///”操作符，这个解决方案也仍是O(*n*)级的。此外，如果把这条语句写成“$str = ~ s/[\Q$remove\E]//g;”，就能对$remove中的巨字符进行转义；可我们不想让这个例子如此复杂难懂。）

这种只用一条简单的语句就能解决问题的事实是有很多人喜欢Perl语言的重要原因，这在文本处理方面体现得更为明显。从另一方面讲，虽然只有一条语句，但不熟悉Perl的程序员是很不容易把握它的，这恰好又是很多人不喜欢Perl语言的理由。如果你属于后一群人，请不要失落——这条语句并不像它看上去那样复杂。“s///”是Perl语言的查找与替换操作符，它的作用是把前两个斜线字符（“/”）之间的内容（在这道例题里，就是$remove中的各个字符）查找出来，并替换为后两个斜线字符之间的内容（在这道例题里，什么也没有，意思就是删除）。最后面的“g”表示对有关内容全部进行替换而非仅替换第一次匹配，“$str = ~”的意思是对字符串$str进行查找与替换。

## 5.5 面试例题：颠倒单词的出现顺序

- **请编写一个函数来颠倒单词在字符串里的出现顺序。比如说，你的函数应该把字符串“Do or do not, there is no try.”转换为“try. no is there not do, or Do”。假设所有单词都以空格为分隔符，标点符号也当做字母来对待。**

你对这道试题可能已经胸有成竹了。既然是对单词进行处理，你的代码就必须识别出单词的起、止位置才行。你可以用一个简单的记号扫描器来遍历字符串中的每一个字符。根据面试题给出的已知条件，你的扫描器必须能够把非单词字符（即空格）与单词字符（除空格以外的其他所有字符）区分开来。单词将以一个单词字符开始，到下一个非单词字符之前或字符串尾结束。

最明显的解决方案是：用记号扫描器来识别单词，把这些单词写到一个临时缓冲区里，最后再把缓冲区里的内容拷贝回原来的字符串。要想颠倒单词在字符串里的出现顺序，你有两条路可选：一是从字符串尾开始做逆向扫描，把识别出来的单词顺序写入缓冲区；二是顺序扫描字符串，但把识别出来的单词从缓冲区尾开始按逆序写入缓冲区。这两条路并没有本质上的区别，在下面的讨论里，我们选择了以逆序扫描字符串的做法。

与往常一样，在动手编写代码之前，你应该先对问题做细致的分析研究。首先，你需要分配一个长度适当的临时缓冲区；然后，进入单词扫描循环，从字符串的最后一个字符开始扫描。如果遇到的是一个非单词字符，你就可以把它直接写到缓冲区里去；但如果遇到的是一个单词字符，你可不能直接把它写到临时缓冲区里去——因为你正在对字符串做逆序扫描，你遇到的第一个字符其实是单词的最后一个字符。如果你按扫描时遇到它们的顺序把字符拷贝到缓冲区里，那么连单词里的字符也将被你给颠倒过来了。你应该继续扫描直到找到单词的第一个字符，然后再把单词里的各个字符按正常顺序拷贝到缓冲区里去。（也许你会认为按正常顺序扫描、按逆序写单词到缓冲区的做法能避免这种复杂性。但在写单词到缓冲区里去的时候，你还是得计算出各单词第一个字符的写开始位置，还是要遇到与与这里类似的问题。）在对单词进行复制的时候，还需要判断出哪个字符是单词的最后一个字符。你可以用检查字符是否是一个单词字符的办法来进行检查，不过，既然你已经知道单词中的最后一个字符的位置，所以一直拷贝到这个位置的办法将更省事。

你可以用一个例子来帮助自己进行分析。假设你用做例子的字符串是“piglet quantum”。你遇到的第一个字符将是“m”。如果你每遇到一个字符就把它拷贝到缓冲区里，你得到的字符串就将是“mutnauq telgip”，这可不是我们预期的结果——“quantum piglet”。要想从“piglet quantum”得到“quantum piglet”，你必须一直扫描到字符“q”，然后再从这个字符开始一直拷贝到字符串的第13个字符“m”，从而得到单词“quantum”。接下来，把空格拷贝到缓冲区里，因为它是一个非单词字

符。再往后，类似与处理单词“quantum”的情况，你识别出“t”是一个单词字符，记下这个位置——字符串的第5个字符——做为单词的尾部标志，然后继续扫描到字符“p”，再把单词“piglet”写到缓冲区里。

最后，当你扫描并复制了整个字符串之后，给临时缓冲区的末尾加上一个NUL字符以表示字符串到此为止；再调用strcpy()把缓冲区里的内容拷贝回原来的字符串。然后，释放临时缓冲区并从函数返回。整个过程如图5-1所示。

很明显，你的扫描器必须在到达字符串的第一个字符后停止扫描。这听起来似乎很简单，但很多粗心大意的人会忘记检查读位置仍在字符串里，特别是当读位置会在函数代码里一个以上的地方发生改变的时候。在这个函数里，你需要1）在主扫描循环里改变读位置以达到下一个记号；2）在单词扫描循环里改变读位置以到达单词的下一个字符。因此，请务必检查这两个循环，确保它们不会跑过了头——超出字符串的第一个字符。

根据以上分析，我们用C语言写出了如下所示的函数代码：

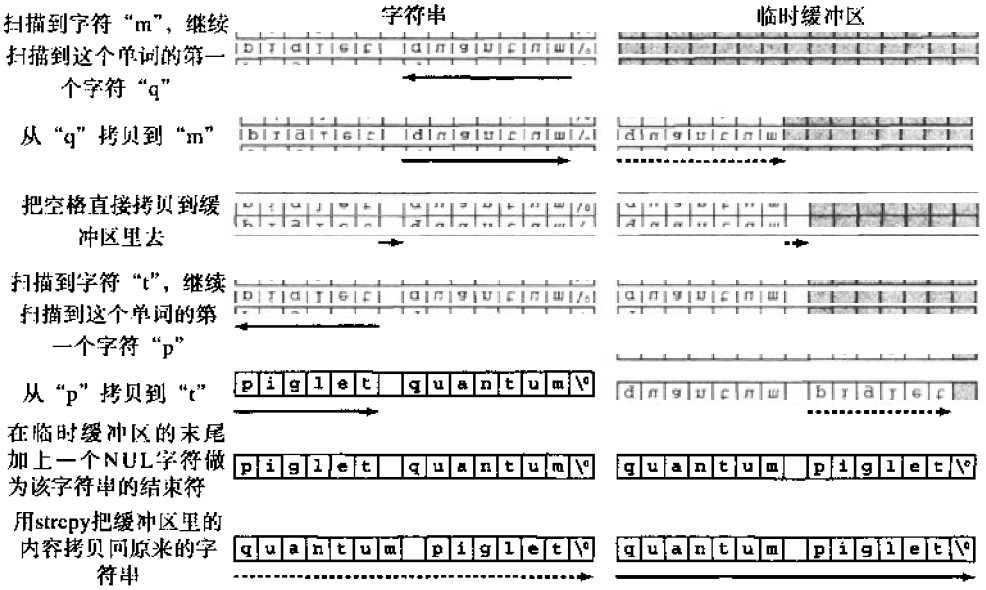



图5-1 颠倒“piglet quantum”。

```
int ReverseWords (char str[])
{
    char *buffer;
    int tokenReadPos, wordReadPos, wordEnd, writePos = 0;

    /* Position of the last character is length - 1 */
    tokenReadPos = strlen(str) - 1;
```

```
    buffer = (char *) malloc(tokenReadPos + 2);
    if (!buffer)
        return 0; /* ReverseWords failed */

    while (tokenReadPos >= 0) {

        if (str[tokenReadPos] == ' ') { /* Non-word characters */

            /* Write character */
            buffer[writePos++] = str[tokenReadPos--];

        } else {  /* Word characters */

            /* Store position of end of word */
            wordEnd = tokenReadPos;

            /* Scan to next non-word character */
            while (tokenReadPos >= 0 && str[tokenReadPos] != ' ')
                tokenReadPos--;

            /* tokenReadPos went past the start of the word */
            wordReadPos = tokenReadPos + 1;

            /* Copy the characters of the word */
            while (wordReadPos <= wordEnd) {
                buffer[writePos++] = str[wordReadPos++];
            }
        }
    }
    /* NUL terminate buffer and copy over str */
    buffer[writePos] = '\0';
    strcpy(str, buffer);

    free(buffer);

    return 1; /* ReverseWords successful */
}
```

这个基于记号扫描器的函数是这类问题的通用性解决方案。它的执行效率和功能扩展性都很不错。是否知道和能否实现这类解决方案是十分重要的；但就这道面试题而言，这一解决方案并不完美。逆向扫描、计算读/写位置、来回拷贝字符/字符串等因素使这个算法显得不那么精巧；需要一个额外的临时缓冲区更让人不满意。

一般说来，程序设计面试题会有一个比较显而易见的通用性解决方案个一个不那么显而易见的专用性算法。与通用性的算法相比，那个专用算法的扩展性要差一些，但它的执行效率会更高，算法也更精巧。就这道面试题来说，我们刚才给出的

是这道面试题的通用性算法，它还有一个专用性的算法。在程序设计面试中，经过分析和比较，你可能会放弃这个通用性的算法而选择那个专用算法。我们之所以要研究这个通用算法并给出其代码实现，是因为记号扫描和字符串扫描是一项重要的程序设计技术；做为一名求职者，你必须知道和掌握它的细节。

为什么要对算法进行改进？为了得到一个执行效率更优的算法。怎样对算法进行改进？把原算法效率低下的地方或者因素找出来并改进之。虽说“精巧“是一个很难量化的衡量标准，但你至少应该努力减少甚至消除原算法对临时性缓冲区的依赖——这有可能会使改进前后的算法出现重大的差异。对算法进行改进并不是件简单的事情。拿这道面试题来说，你不能简单地把算法修改成同时对原字符串进行读、写的样子——否则，你将会把尚未被读出的字符覆盖掉。

你与其浪费时间去考虑在没有了临时缓冲区后不能做些什么，还不如把注意力集中到你现在还能做些什么的方面。比如说，用交换字符的办法也能对字符串进行颠倒。我们先找个例子来试一下：通过加换字符，字符串“in search of a algorithmic elegance”将被转换为“ecnagele cimhtirogla fo hcraes ni”。请再仔细看看这两个字符串！单词已经被排列为正确的顺序，只是单词中的字符顺序是颠倒的。你只要把这个被颠倒过来的字符串中的每一个单词再颠倒一遍，就能得到你想要的东西了。你可以参考刚才那个算法的思路用一个扫描器来判断单词的第一个字符和最后一个字符，再通过一个用来颠倒字符串的函数把每一个单词（也就是一个子字符串）再颠倒一遍。

现在，用不着再分配临时性的缓冲区了，你只需编写一个对字符串本身进行颠倒的函数就能完成任务。但要注意一点：在C语言里，你无法只用一条语句就交换两个值——你必须使用一个临时变量和三条赋值语句才能完成这一交换。那个用来颠倒字符串的函数需要有三个输入参数：一个字符串、一个起点下标、一个终点下标。先交换位于起点和终点处的那两个字符，再递增起点下标和递减终点下标，然后循环。当起点下标和终点下标相逢在中点（当字符串的长度是奇数个字符时）或起点下标大于终点下标（当字符串的长度是偶数个字符时）的时候，循环结束，字符串也就被颠倒过来了——在此之前，起点下标将一直小于终点下标，你的循环语句也就得继续循环下去。

下面是用C语言写出来的单词颠倒函数，它还需要调用一个字符串颠倒函数：

```
void ReverseWords(char str[])
{
    int start = 0, end = 0, length;

    length = strlen(str);

    /* Reverse entire string */
```

```
    ReverseString(str, start, length - 1);

    while (end < length) {
        if (str[end] != ' ') { /* Skip non-word characters */

            /* Save position of beginning of word */
            start = end;

            /* Scan to next non-word character */
            while (end < length && str[end] != ' ')
                end++;

            /* Back up to end of word */
            end--;

            /* Reverse word */
            ReverseString(str, start, end);
        }
        end++; /* Advance to next token */

    }
    return;
}

void ReverseString (char str[], int start, int end) {
    char temp;
    while (end > start) {
        /* Exchange characters */
        temp = str[start];
        str[start] = str[end];
        str[end] = temp;

        /* Move indices towards middle */
        start++; end--;
    }
    return;
}
```

与前面给出的那个通用性算法相比，这个专用性的算法不需要使用临时性的缓冲区。它不仅更加精巧，在执行效率上也有了很大提高——它不仅消除了动态内存分配方面的开销，也不需要再来回拷贝字符和字符串了。

最后，做为对比，我们还想把完成同一任务的Perl语言代码介绍给大家。我们用了这么多篇幅才完成其算法分析并写出其代码的C语言程序在Perl语言里只需一条语句就搞定了：

```
$ReversedWords = join(" ", reverse(split(/ /, $Words)));
```

其中，split负责把字符串$Words按空格分断为一个单词列表，reverse负责颠倒

这个单词列表中的单词顺序，最后再由join用空格把顺序颠倒后的单词列表合并成一个字符串。与那么多的C语言代码相比，这条Perl语句写起来是既简单又迅速，但在经过解释（和编译）之后，Perl语言执行代码的效率却要比C语言执行代码的差很多。总而言之，Perl是一种能提高程序员工作效率的高级语言，而C则是一种能提高程序执行效率的低级语言，它们的优、缺点在这道面试题上表现得淋漓尽致。

## 5.6 面试例题：整数/字符串之间的转换

- **请编写一个函数，它的调用接口如下所示。第一个函数负责把一个ASCII字符串转换为一个带符号整数，第一个函数负责把一个带符号整数转换为一个ASCII字符串。**

```
int StrToInt(char str[]);
void IntToStr(int num, char str[]);
```

**已知条件：（1）传递给IntToStr函数的缓冲区的长度足以容纳int整数范围内的任何一个数；（2）传递给StrToInt的字符串只包含数字和“-”（负号）；也就是说，它代表着一个格式正确的整数值，并且落在int整数的范围内。**

在C语言里，整数与字符串之间的转换通常是用库函数ssanf()和sprintf()来完成的。你应该向面试考官说明这一点并指出这样一个事实：在正常情况下，没有必要用自己写的代码去实现标准函数库已经提供了的功能。不过，这番话并不能免除你的“痛苦”——你还是得老老实实地把试题要求的函数写出来。

我们先来实现StrToInt函数。这个函数的输入参数是一个合法整数的字符串表示形式。想想看，这一点能让你做些什么？假设你有一个字符串“137”。这是一个由三个字符组成的字符串，在位置0处是一个代表着字符“1”的ASCII值，在位置1处是一个代表着字符“3”的ASCII值，在位置2处是一个代表着字符“7”的ASCII值。我们在中学就已经学过，那个“1”代表着100，因为它是一个百位数；那个“3”代表着30，因为它是一个十位数；而那个“7”代表的就是7，因为它是一个个位数。把这几个值加在一起将得到一个数字：100 + 30 + 7 = 137。

分解一个整数的字符串表示形式并生成相应整数值的思路就由此而产生。你需要确定字符串中的每一个字符所代表的（整数）数字，把这个数字与相应的位数值相乘，再把各部分的积加起来。

我们先来解决ASCII字符到数字的转换问题。数字字符的ASCII值有什么特点呢？它们是顺序排列的：字符“0”的ASCII值加上1等于字符“1”的ASCII值，字符“1”的ASCII值加上1等于字符“2”的ASCII值，字符“2”的ASCII值加上1等于字符“3”的ASCII值，依次类推。（要是你连这个都不知道，那就只能向面试考官求助了。）因此，数字字符的ASCII值就等于这个数字加上字符“0”的ASCII值。（注意，

字符“0”的ASCII值是一个不等于零的数值，而这个数值代表着字符“0”。）也就是说，你只要用数字字符的ASCII值减去字符“0”的ASCII值就能得到该数字的整数表示形式。也许你想不起字符“0”的ASCII值到底是多少，但编译器知道——你只要写出代码“- '0'”，编译器就会把它解释为“减去字符串‘0’的ASCII值”。

接下来，你还需要知道每个数字所在的位数以正确地进行乘法计算。在知道那个整数的长度之前，你是无法知道它第一个数字的位数到底是多少的，所以按从左向右的顺序来处理这些数字好像有点麻烦。比如说，“367”的第一个数字与“31”的第一个数字是一样的，但前者代表着300，而后者只代表30。这样看来，按从右向左的顺序来扫描数字将是最好的办法，因为最右边的数字永远是一个个位数，而倒数第二个数字则永远是一个十位数，依次类推。也就是说，字符串的最右端（即字符串尾）对应着个位，位数值是1；每往左偏移一位，相应的位数值就要乘上一个10。这个方案好是好，可每次循环都不得不做两次乘法才行：一次是把某位置上的数字与相应的位数值相乘；另一次是为了求出下一个位置的位数值。这样做的效率好像不怎么高呀。

我们是不是把按从左向右的顺序来扫描字符串的方案排除得太轻率了？按照刚才的分析，在扫描完整个字符串之前，你是无法确定某个数字的位数值的。有没有能绕过这个问题的办法呢？我们再仔细研究一下字符串“367”的例子。当你遇到第一个字符——“3”——的时候，你知道它的值是3。如果下一个字符是字符串尾，那么对应的整数值就将是3。可你又遇到了字符“6”，于是，字符“3”将代表30，而字符“6”就代表着6。在下一次循环里，你遇到了最后一个数字——“7”，于是，字符“3”将代表300，字符“6”将代表60，字符“7”将代表7。看出什么了吗？每新遇到一个字符，已经被扫描过的字符所对应的位数值就都要乘上一个10。不管你刚开始时遇到的“3”代表的是3、30还是30000，只要你新遇到一个字符，就需要把此前计算出来的数值乘以10，然后再加上你新扫描出来的那个数字。你根本用不着去计算那些位数值，换句话说，这个方案能在每次循环里省略一次乘法运算。这个优化技巧很重要，在需要计算校验和（checksum）的场合里经常会用到；这个聪明的办法还有一个专门的名字：Horner法则。

到目前为止，我们的研究还仅限于正整数。怎样把这个方案扩展到负整数上去呢？负数的第一个位置上有一个负号（“-”），这个字符不能转换为一个数字，所以你必须跳过它；可你也不能不对它进行处理。在扫描完整个字符串并计算出最终的数值之后，你还需要改变这个数值的符号，好让它成为一个负数。你可以用乘上一个“-1”的办法改变数值的正负号。也就是说，在扫描字符串的时候，你必须先检查它的第一个字符是不是一个负号。如果是，你就得跳过这个字符；然后，等你扫描出所有的数字并计算出与之对应的整数值时，再乘上一个“-1”来改变那个数值的正负号。具体做法是：如果字符串的第一个字符是“-”，就设置一个标志；等计

算出最后结果时，就根据这个标志决定是否还需要再乘上一个“-1”。

我们把对这个算法的分析总结如下：

```
把整数值初始化为0
如果字符串的第一个字符是“-”
    设置负数标志
    从第二个字符开始进行扫描
对字符串中的每一个字符
    把整数值乘上一个10
    把（数字字符-'0'）与整数值相加
返回整数值
```

用C语言写出来的函数代码如下所示：

```
int StrToInt(char str[])
{
    int i = 0, isNeg = 0, num = 0;

    if (str[0] == '-') {
        isNeg = 1;
        i = 1;
    }

    while (str[i]) {
        num *= 10;
        num += (str[i++] - '0');
    }

    if (isNeg)
        num *= -1;

    return num;
}
```

在把这个函数提交给面试考官之前，你还应该先检查一下它是否对有关的特例情况进行了适当的处理。为了确保这个函数能够对整数、负数和零做出正确的处理，你至少应该用-1、0、1来对它进行一下检查。你还应该用一个多位数（比如“324”）来进行一次检查，以保证循环语句部分不会出现问题。还好，这个函数在这几种情况下都没有出现问题；你可以转移到下一个目标——IntToStr函数。

IntToStr函数的工作与StrToInt函数的正好相反。既然如此，我们在编写StrToInt函数之前所做的分析就差不多都能在这里用上。比如说，既然用数字字符减去字符“0”的ASCII值能得到与之对应的数字，那么给数字加上字符“0”的ASCII值就应该得到与之对应的数字字符（也就是把数字转换为字符）了。

在把数字转换为ASCII值之前，你需要知道这些数字到底是多少。怎样才能做到这一点呢？假设你手里有个整数值732。如果把这个数字写在纸上，你将毫无困难地看出它的三个组成数字是7、3、2。可是，计算机使用的并不认识十进制数字，它们只认识二进制数值1011011100。因为你无法从一个二进制数里直接“看出”它的十进制组成数字，所以你必须通过一些计算才能完成这一任务；而确定这些十进制组成数字的思路不外乎两种：从左向右或从右向左。

先来看看从左向右的方案。如果用100（百位数的位数值）对732做整数除法，你将得到第一个数字——7；如果用10（十位数的位数值）对732做整数除法，你将得到73而不是3。这么看来，你得先减去百位数的值才能继续去找十位数。于是，你将得到下面这一连串的结果：

$732 \div 100 = 7$（第一个数字）；$732 - 7 \times 100 = 32$

$32 \div 10 = 3$（第二个数字）；$32 - 3 \times 10 = 2$

$2 \div 1 = 2$（第三个数字）

要想实现这个算法，你必须先知道第一个数字的位数值，然后每找到一个新数字，就把位数值除以一个10。这个算法行是行，就是复杂了点儿。那么，从右向左的方案怎么样？

还是用732做例子。怎样才能计算出2这个最右边的数字呢？732除以10的余数是2（C语言中的整数求余操作符是“%”）。怎样才能求出下一个数字呢？732除以100的余数是32，再用10对32做整数除法将得到3。但此时你已经要用到两个位数值（100和10）了；要是这个数字再长一点，岂不是更麻烦？

如果“先做整数除法、再做求余计算”会怎么样？用10对732做整数除法得73，73除以10的余数是3。再往后，73 / 10 = 7；7 % 10 =7。这个方案好像很不错——你用不着关心那些位数值，只要反复求出整数除法的商和余数，直到没有东西剩下来为止就行了。

这个办法的主要缺陷为你是按逆序确定出各个数字的。在找出全部数字之前，你无法得知它们到底有多少个，因而也就无法确定与之对应的字符串的长度，也就无法知道应该从什么地方开始写出这个字符串来。你可以用计算两遍的办法来解决这一问题：一遍用来确定数字的个数，好让自己知道该从什么地方开始写出它们来；一遍用来真正地写出这些数字。可这多少有些拖泥带水的味道，能不能更干净利落地解决这一问题呢？我们想到的更好的解决方案是：先按逆序写出这些数字，然后再对字符串做一次颠倒操作。你可以直接用输出字符串来完成这些操作，也可以先把这些数字写到一个临时缓冲区里（因为整数的最大取值所对应的字符串也很短），然后再把它们颠倒过来写入最终的字符串。

到目前为止，我们还没有涉及负数的问题。这个基于整数除法（商和余数）的方案不适用于处理负数，因为 -32 % 10 = 8，不等于 -2或2。不过，大家也许已经想到

了，你可以先检查这个数字是不是一个负数并对各个数字做出调整。但这未免有些复杂和效率低下。在StrToInt函数里，我们把负数当做正数来处理，然后在最后根据一个负数标志做一次转换。这个办法能不能用在这里呢？你可以先判断那个正数是不是一个负数，如果是，就给它乘上一个“-1”并设置负数标志；最后再根据负数标志在结果字符串的开头写上一个负号“-”。与调整每一个数字相比，这个办法可简单多了——你只要别忘了在乘完“-1”之后设置好负数标志就应该不会有问题了。

在解决了IntToStr函数所有这些重要的子问题后，你只要把这些步骤拼凑起来就能组成一个代码大纲了：

```
如果整数值小于0
    给它乘上一个“-1”
    设置负数标志
当整数值不为零时，循环
    （整数值 % 10）+'0'，然后把得到的ASCII值写入临时缓冲区
    用10对整数值做整数除法
如果负数标志被置位
    在临时缓冲区的下一个位置写入一个负号“-”
把临时缓冲区里的字符按逆序写入输出字符串
在输出字符串的末尾加上一个NUL字符
```

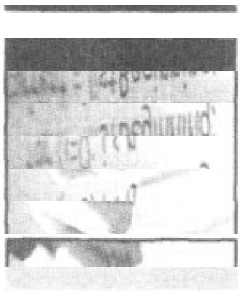
用C语言实现的IntToStr函数完整代码如下所示：

```
#define MAX_DIGITS_INT 10
void IntToStr(int num, char str[])
{
    int i = 0, j = 0, isNeg = 0;
    /* Buffer big enough for largest int, - sign and NUL */
    char temp[MAX_DIGITS_INT + 2];

    /* Check to see if the number is negative */
    if (num < 0) {
        num *= -1;
        isNeg = 1;
    }
   /* Fill buffer with digit characters in reverse order */
   while (num) {
       temp[i++] = (num % 10) + '0';
       num /= 10;
   }

   if (isNeg)
       temp[i++] = '-';
```

```
    /* Reverse the characters */
    while (i > 0)
        str[j++] = temp[--i];

    /* NUL terminate the string */
    str[j] = '\0';
}
```

现在，用检查StrToInt函数时使用的特例（多位数、－1、0、1）来验证IntToStr函数。多位数、－1、1都没有问题；但当num等于0时，程序将不会进入while循环体。这将使IntToStr函数写出一个空字符串而不是字符“0”。怎样消除这个bug呢？你必须让程序至少进入while循环体一次，这样才能在num以0开始的时候往临时缓冲区里写上一个“0”，进而让结果字符串里也有这个“0”。把while循环改为do...while循环就能确保循环体至少被执行一次。改动后的代码如下所示，它对0、整数、负数等情况都能做出正确的处理了。

```
#define MAX_DIGITS_INT 10
void IntToStr(int num, char str[])
{
    int i = 0, j = 0, isNeg = 0;
    /* Buffer big enough for largest int, - sign and NUL */
    char temp[MAX_DIGITS_INT + 2];

    /* Check to see if the number is negative */
    if (num < 0) {
        num *= -1;
        isNeg = 1;
    }

    /* Fill buffer with digit characters in reverse order */
    do {
        temp[i++] = (num % 10) + '0';
        num /= 10;
    } while (num);

    if (isNeg)
        temp[i++] = '-';

    /* Reverse the characters */
    while (i > 0)
        str[j++] = temp[--i];

    /* NUL terminate the string */
    str[j] = '\0';
}
```



# 递归算法

递归本身是一个很简单的概念：任何一个对自己进行调用的函数或例程都是递归的。但这个简单的概念在理解和运用中却往往相当复杂。掌握递归技术的难点之一是有关递归技术的讨论都非常理论化，不仅很抽象，还需要具备足够的数学知识。虽然理论方面的探讨有很高的价值，但本章还是打算采用一种更“原始”的办法，即通过一些例子和实际应用来比较递归算法与循环算法（非递归的）的异同。

递归技术特别适合用来解决嵌套着类似的子任务的任务。比如说，排序、搜索、遍历等方面的问题通常都会有一个比较简单的递归性解决方案。递归算法通过逐步解决嵌套着的子问题来达到最终解决整个问题的目的。编写正确的递归算法迟早会到达一个不需要再进一步调用其自身就能解决的子任务。这个情况，即递归函数到达不再需要做进一步递归时的情况，叫做递归算法的“基底情况”（base case）；而此前需要做进一步递归的各种情况都叫做“递归情况”（recursive case）。

**提示：** *递归算法应该包括两种情况：递归情况和基底情况。*

这些概念可以用一个简单而又常用的实际例子——即整数的阶乘——来说明。$n!$（$n$阶乘）等于从$n$到1之间的所有整数的乘积。比如说，$4! = 4 \times 3 \times 2 \times 1 = 24$。$n!$的数学定义是：

$$n! = n(n-1)!$$

$$0! = 1! = 1$$

根据这一定义，我们将轻而易举地得到一个计算阶乘的递归性算法。我们的任务是计算出$n!$的数值，而相应的子任务则是计算出$(n-1)!$的数值。在递归情况（即$n$大于1时的情况）里，函数将调用它自己去计算$(n-1)!$与$n$的乘积；在基底情况（即$n$等于0或1时）里，函数简单地返回数值1就行了。把这个算法用C语言写出来，我们

将得到如下所示的代码：

```
int Factorial(int n) {
    if (n > 1) {  /* Recursive case */
        return Factorial(n-1) * n;
    }
    else {        /* Base case */
        return 1;
    }
}
```

图6-1给出了这个函数在计算4!时的执行流程。请注意，函数每递归调用一次，$n$的值就会减去1。这将确保基底情况迟早会被遇到。如果递归函数编写得不正确，就可能出现无法到达基底的情况，它将陷入无限递归的境地。不过，无限递归只是一种理论情况，很少会出现在实际工作中——迟早会因为堆栈溢出而使程序崩溃；崩溃也算是一种停止，只不过是一种灾难性的停止罢了。（注意：有一种称为“尾递归”（tail recursion）的递归形式，这种递归在经过编译器的优化后，每次递归都将使用同一个堆栈结构。如果编写得不好，尾递归在优化后是有可能出现无限递归的情况，因为它不会造成堆栈溢出。）

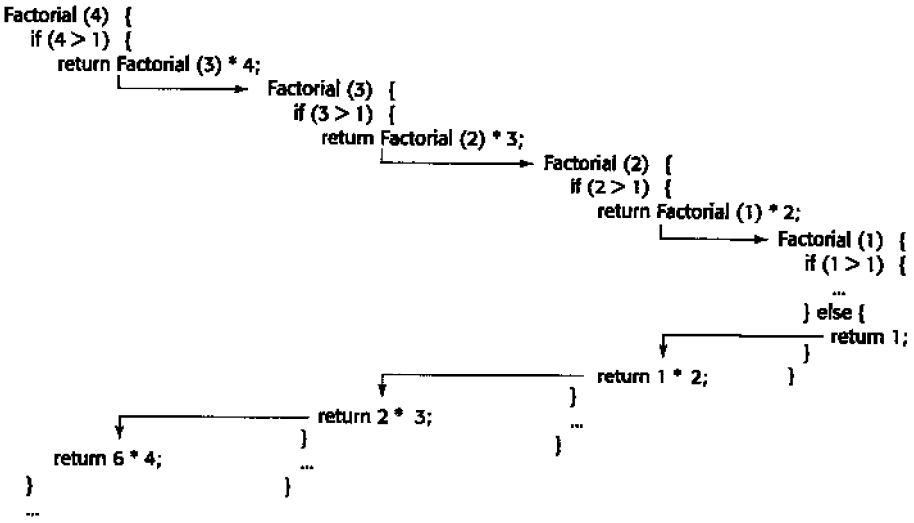



图6-1　4!的计算过程。

**提示：** 递归情况演变到最后必须到达一个基底。

计算阶乘只是递归函数最简单的示例之一。在很多场合，你的递归函数可能需要额外的数据结构或参数来跟踪递归调用的嵌套层次。在遇到这类情况的时候，最

好是把对有关数据结构和参数进行初始化的代码放到另外一个函数里去，由这个“包装”函数在完成有关数据的初始化工作后再去调用那个纯粹的递归函数，从而给程序的其他部分提供一个简洁的调用接口。

比如说，假设你需要返回阶乘计算函数的全部中间结果（即小于$n$的整数的阶乘值）和最终结果（即$n!$）。最自然的做法莫过于把这些计算结果放到一个整数数组里去，这意味着你的函数需要分配一个数组。你还需要知道各计算结果将被写到数组的哪一个位置里。这些工作可以用一个包装函数轻松地完成，如下所示：

```
int *AllFactorials(int n) /* Wrapper function */
{
    int *results;
    int length = (n == 0 ? 1 : n);
    results = (int *) malloc(sizeof(int) * length);
    if (!results)
        return NULL;
    DoAllFactorials(n, results, 0);
    return results;
}

int DoAllFactorials(int n, int *results, int level)
{
    if (n > 1) {  /* Recursive case */
        results[level] = n * DoAllFactorials(n - 1, results, level + 1);
        return results[level];
    }
    else {        /* Base case */
        results[level] = 1;
        return 1;
    }
}
```

请看，这个包装函数把内存分配和记录嵌套层次的细节都“收拾”得干净利落，我们的递归函数只需专心完成计算工作就行了。就这个例子来说，利用变量$n$来确定数组下标也是可行的，这将节省一个level变量。但很多递归函数却只能通过增加一个变量的办法来跟踪嵌套层次。

**提示：** 如果递归函数本身会比较复杂，就应该考虑用另外一个函数来对数据进行初始化等工作。

递归技术能够方便地解决很多问题，但它也许并非最佳解决方案，我们也很难说它是最有效率的解决方案。这是因为递归算法需要进行大量的函数调用，这在任何一种计算机平台上都是一项开销巨大的活动。即使是一个像计算整数阶乘这样简单的递归函数，大多数计算机平台用来处理函数调用的时间也要远多于用来进行实

际计算的时间。用循环语句来代替递归调用的遍历算法往往会有更高的执行效率，因为它不需要如此巨大的开销。

**提示：** 循环算法往往要比递归算法的执行效率更高。

只要是能够用递归算法解决的问题，也就一定能用循环语句来解决。即使是一些比较“纯粹”的递归性任务，它们的循环算法也不见得很难写。比如说，用循环语句来计算整数阶乘的代码就很简单。展开阶乘的定义公式，我们就能更清楚地看出，它其实只是$n$到1之间的全体整数（包括$n$和1在内）的连续乘积。所以我们完全可以用一个for循环来计算出这个值来。下面是用来计算阶乘的C语言循环算法：

```
int Factorial(int i) {
    int n, val = 1;
    for (n = i; n > 1; n--)  /* n of 0 or 1 fall through */
        val *= n;
    return val;
}
```

与前面给出的递归算法相比，这个算法的执行效率有了很大的提高。在速度测试中，它的运行速度几乎是递归算法的三倍。虽说两种算法在解决这一问题上的思路不同，但非递归算法并不比递归算法难写多少。

有些问题没有明显的循环算法。即便如此，我们也完全能够实现出一个不使用递归函数的递归算法来。递归调用的主要作用是保存局部变量的当前值；这样，当某嵌套层次上的递归调用完成了该层次上的子任务并返回时，这些局部变量的值就能得到恢复，使程序流程一层一层地返回到出发点。局部变量都保存在程序堆栈上，所以递归函数的每一次嵌套调用都会有它自己的一组局部变量。也就是说，递归函数在调用时会隐含地把变量的值保存到程序堆栈上去。如果为函数明确地分配一个它自己的堆栈，再让函数自己去负责完成在这个堆栈上保存或检索局部变量值的操作，就用不着再对函数进行递归调用了。不过，这种“改进”算法往往要比对应的递归算法复杂得多，也难以实现得多。如果堆栈方面的开销并不比递归函数调用的开销小多少，循环算法的执行效率就未必能比递归算法的执行效率高出多少来。如果大幅增加的复杂性在执行效率方面得不到满意的回报，那么还是用递归函数来实现一个递归算法比较好——除非你有不得已的理由，比如参加程序设计面试等。没有用递归函数而实现的递归算法我们前面已经见过了，第4章“树和图”中的面试例题“左遍历，不使用递归”的解决方案就是一个很好的例子。

**提示：** 递归算法可以不使用递归函数来实现。你可以采用堆栈来解决同样的问题，但这样做往往得不偿失。

在程序设计面试中，一个能够完成任务的解决方案是最重要的；解决方案的执

行效率应该放在第二位考虑。因此，除非试题另有要求，你应该从最先想到的解决方案入手。如果它碰巧是一个递归性的方案，你不妨先向面试考官说明一下递归算法天生的低执行效率问题——表明你知道这些事情。有时候，如果你同时想到了两个解决方案，其一是递归性的，其二是循环性的，并且它们的复杂程度相差不多，你应该把这两个方案都向面试考官介绍一下，然后告诉他们说你准备实现哪个循环性的算法——因为它的执行效率可能会更好一些。

## 6.1 面试例题：二分法搜索

- **请实现一个函数，用它来对一个排好序的整数数组进行二分法搜索（binary search）。这个函数的调用模型应该是下面这个样子：**

```
int BinarySearch(int* array, int lower, int upper, int target);
```

**请对这个搜索算法的执行效率进行评估，并把它与其他搜索方法进行比较。**

所谓二分法搜索说的是这样一种搜索方法：把搜索目标与搜索区间（比如这道试题中的数组）二分之一位置处元素进行比较，如果后者小于前者，则不再对搜索区间前半部分中的其他元素进行比较；如果后者大于前者，则不再对搜索区间后半部分中的其他元素进行比较；如果两者恰好相等，说明你已经找到了你想要的东西，就可以停止这次搜索了。重复上述过程，直到找到目标元素或者搜索完整个搜索区间为止。即使你已经把老师在计算机科学课程里讲过的东西忘得差不多了，这个算法也应该让你想起一个小时候玩的游戏：几个孩子在猜数字，一个孩子猜，另一个孩子则回答对方说他猜的是“高”还是“低”；二分法是玩好这个游戏的最佳策略。

因为二分法搜索呈现出一种递归的特性——反复地对剩下来的搜索区间进行二分法搜索，所以人们立刻会想到采用递归技术来实现其解决方案。你的函数需要以下几个输入参数：一个将被分的数组、搜索区间首尾两端的下标、做为搜索键字的元素值。用搜索区间的上限减去它的下限将得到这个搜索区间的长度，再把这个值除以2并加上搜索区间的下限就得到了搜索区间中央元素的下标值。接下来，把该元素与搜索目标元素进行比较，如果它们相等，返回中央元素的下标值；如果目标元素比较小，则以“中央元素的下标减去1”做为新的搜索区间上限；如果目标元素比较大，则以“中央元素的下标加上1”做为新的搜索区间下限。重复这一递归过程直到你找到一组匹配为止。

在动手编写代码之前，你应该先把可能会遇到的出错情况考虑周全。在考虑这个问题的时候，你应该从与搜索区间中的数据有关的已知条件入手，看有哪些情况会导致程序运行结果与这些已知条件冲突。这道面试题的已知条件告诉我们，你将对一个排好序的数组进行二分法搜索；所以必须检查你遇到的数组是不是排好了序的。你可以用比较搜索区间的上限元素是否小于下限元素的办法来进行这项检查，如果是，就应

该返回一个出错代码。另一方案是先调用一个排序例程，然后再次进行搜索；不过，因为这是在参加程序设计面试，所以你大可不必这样做。另外，你的搜索需要假定目标元素确实存在于那个数组当中。如果你只在找到这个元素后才停止递归，那么，万一出现数组里没有目标元素的情况，你的函数就可能会陷入无限递归。这种情况可以这样来解决：如果上限下目标和下限下目标相等但该位置上的元素不是你要搜索的元素时，就返回一个出错代码。最后，下限下标应该小于或等于上限下标。如果不是这样，简单的办法是返回一个出错代码；但如果你是在编写一个真正的程序而非参加程序设计面试，就应该选择下面两种做法之一：1）把这种情况定义为一个非法调用并用一个验证（assert）操作来检查它，这个办法偏重于追求程序的执行效率；2）默默地颠倒搜索区间的上、下限，这个办法的代码比较容易编写。

现在，我们把这些算法和出错检查机制编写成代码：

```
#define E_TARGET_NOT_IN_ARRAY -1
#define E_ARRAY_UNORDERED -2
#define E_LIMITS_REVERSED -3

int BinarySearch(int* array, int lower, int upper, int target)
{
    int center, range;

    range = upper - lower;
    if (range < 0) {
        return E_LIMITS_REVERSED;
    } else if (range == 0 && array[lower] != target) {
        return E_TARGET_NOT_IN_ARRAY;
    }

    if (array[lower] > array[upper])
        return E_ARRAY_UNORDERED;

    center = ((range)/2) + lower;

    if (target == array[center]) {
        return center;
    } else if (target < array[center]) {
        return BinarySearch(array, lower, center - 1, target);
    } else {
        return BinarySearch(array, center + 1, upper, target);
    }
}
```

这段代码能够完成面试题要求的任务，但它的执行效率却不一定让人满意。正如我们在本章开篇部分中讨论的那样，递归算法的执行效率通常比不上功能相当的

循环算法。

仔细研究一下上面的函数代码就会发现，每次递归调用都只是改变搜索区间的上、下限而已。你完全可以用一个循环语句来改变搜索区间的上、下限，从而消除掉因递归调用的增加带来的开销。如下所示：

```
int IterBinarySearch(int* array, int lower, int upper, int target)
{
    int center, range;

    if (lower > upper)
        return E_LIMITS_REVERSED;

    while (1) {
        range = upper - lower;
        if (range == 0 && array[lower] != target)
            return E_TARGET_NOT_IN_ARRAY;

        if (array[lower] > array[upper])
            return E_ARRAY_UNORDERED;

        center = ((range)/2) + lower;

        if (target == array[center]) {
            return center;
        } else if (target < array[center]) {
            upper = center - 1;
        } else {
            lower = center + 1;
        }
    }
}
```

因为每次递归（或循环）都会筛选掉一半的元素，所以二分法搜索的执行时间是O(log(*n*))。这比用从头至尾顺序扫描数组的办法——它的执行时间是O(*n*)——执行效率要高出不少。不过，要想进行二分法搜索，你必须先对数组进行排序，排序操作的执行时间一般是O(*n* log(*n*))。

## 6.2 面试例题：字符串的全排列

- **请编写一个函数，用它把字符串中所有字符的各种排列形式全都显示出来；换句话说，用给定字符串里的字符做全排列。比如说，如果给定字符串是“hat”，你的函数就必须输出字符串“tha”、“aht”、“tah”、“ath”、“hta”、和“hat”。字符串中的每一个字符都是彼此不相干的，即使有重复出现字符，也要把它们当做不同的东西。比如说，如果给定字符串是“aaa”，你的函数就必须输出6**

**个字符串“aaa”。你可以以任意顺序来输出字符串全排列的结果。**

以人工方式来写出一个字符串的全排列是一件非常简单的事情，可为这个过程描述出一个算法可就有点困难了。这就像是让你描述一下你是如何系鞋带的：你早就知道答案，但要把其中的步骤准确地描述出来，你可能还是要反复多实际操练几次才行。

为了解答这道面试题，你就实际操练几次吧：先以人工方式对一个短字符串进行一下全排列，然后逆向推导出一个能够重现这一过程的算法来。我们使用的例子是字符串“abcd”。既然是为一个简单的过程构造一个算法，那我们当然要按一种系统化的顺序来进行。你选用的系统化顺序到底是什么样的并不重要——不同的顺序将导致不同的算法，可只要你是按一定顺序来完成这项工作的，就肯定能推导出一个算法来。不过，比较简单的顺序能够使这项工作变得更容易一些，也更能避免你漏掉一两种排列。

你可以按字母表顺序来列出这些字符排列。这就意味着第一组排列将全部是以字符“a”打头的。在这组排列里，你先以字符“b”做为第二个字符来写出各种可能的排列，再依次以字符“b”、“c”、和“d”做为第二个字符来写出各种可能的排列。如此这般得到的字符串全排列将如下所示：

| | | | |
|---|---|---|---|
| abcd | bacd | cadb | dabc |
| abdc | badc | cadb | dacb |
| acbd | bcad | cbad | dbac |
| acdb | bcda | cbda | dbca |
| adbc | bdac | cdab | dcab |
| adcb | bdca | cdba | dcba |

在继续往下之前，请检查一下你是否遗漏了某些种排列。可能出现在第一位置上的字符有四种，可能出现在第二位置上的字符有三种（因为第一位置已经用掉了一个字符），可能出现在第三位置上的字符有两种（因为第一、第二位置已经用掉了两个字符），可能出现在第四位置上的字符就只剩下一种了（因为第一、第二、第三位置已经用掉了两个字符）。$4 \times 3 \times 2 \times 1 = 24$，所以你将总共得到24种不同的排列。上面列出来的就是24种，所以你什么也没有遗漏。你是不是觉得这个算法很面熟？没错，它就是4的阶乘（4!）——想起来了吗？$n!$就是$n$个物体的全排列种数。

我们再来分析一下这些排列的规律。最右边的字符要比最左边的字符变化快。在写下第一个（最左边的）字符后，把以它打头的各种排列全都写出来，然后把第一个字符换一下。类似地，在写下第二个字符后，把相应的各种排列全都写出来，然后把第一或第二个字符换一下。换句话说，你可以对全排列过程做出如下定义：为当前位置选择一个字符，向右移动一个位置，再以新位置为当前位置继续进行全排列。这听起来像是全排列操作的一个递归性定义。我们再用递归语言把这一过程描述得更清楚

一些：为了找出以位置$n$为起点的全排列，需要把允许出现在位置$n$处的字符依次放到这个位置上，然后为位置$n$上的每一种新字符找出以位置$n+1$为起点的全排列来（递归情况）；当$n$大于输入字符串中的字符个数时，就说明你已经完成了全排列——此时，先把当前的字符排列打印出来，再返回到$n-1$位置去更改字母（基底情况）。

这个算法只差一点就全齐备了；你还得把“允许出现在位置$n$上的字符”定义得更严密一些。因为输入字符串中的每一个字符只能在一种排列里出现一次，所以“允许出现在位置$n$上的字符”肯定不能是输入字符串里的全体字符。再仔细回忆一下你以人工方式写出字符串全排列的过程。对以字符“b”打头的全排列来说，“b”只能出现在第一位置，不可能再次出现在其他位置上——因为你想挑选第一个字符的时候，“b”早已经有位置了。对以字符“bc”打头的全排列来说，第三和第四位置上的字符只能是“a”和“d”各占一个——因为“b”和“c”已经有位置了。可见，所谓“允许出现在位置$n$上的字符”其实是输入字符串中尚未被当前位置左侧（即编号小于$n$）的各个位置所选用的字符。因此，你可以把位置$n$上的每一个候选字符与它前面各位置上已经被选用的各个字符进行比较，看它是否已经被用过了。这种低效率的扫描操作可以用这个办法来消除：分配一个布尔值数组，让它的每个元素与输入字符串中的一个字符相对应；根据各字符是否已被选用的情况来把对应的数组元素设置为“已使用”或“未使用”。

这个算法的总结性描述如下所示：

```
如果你已经超过了最后一个位置
    打印当前字符串
    返回
否则
    对输入字符串中的每一个字符，循环
      如果它已被标记为“已使用”，跳到下一个字符
      否则，把这个字符放到当前位置上
        把这个字符标记为“已使用”
        从下一个位置开始用剩下来的字符做全排列
        把这个字符标记为“未使用”
```

把递归算法细分为基底情况和递归情况是一种良好的程序设计方法，这将使代码更容易阅读和理解，但这种做法不能提供最优的执行效率。有一种大幅提高执行效率的办法：如果下次递归将到达基底情况，就不再进行递归调用，而是直接进入基底情况。就这个算法来说，这意味着你要检查放在当前位置上的字符是不是最后一个字符——如果是，就不进行递归调用而直接打印出这种排列；如果不是，才进行下一次递归调用。这种优化办法能够减少$n!$函数调用，把函数调用减少$n$倍（$n$是输入

字符串的长度）。人们把这种直接进入基底情况的优化方法称为“断臂递归”（arms length recurision），并认为这是一种不好的程序设计方法——尤其是在学术圈里。不管你是否采用了这种优化办法，向面试考官解释一下其中的利弊都是很有必要的。

为了换换方法，我们将用Perl来实现这个算法，如下所示：

```
sub Permute {
    my ($inString) = @_;
    # $in, $out and $used are references to arrays
    my ($in, $out, $used) = ([], [], []);
    @$in = split //, $inString;  # One char in each element of $in
    DoPermute($in, $out, $used, 0);
}

sub DoPermute {
    my ($in, $out, $used, $recursLev) = @_;
    my ($i);

    # Base case
    if ($recursLev == @$in) {
        print @$out, "\n";
        return;
    }

    # Recursive case
    for ($i = 0; $i < @$in; $i++) { # @$in gives array length
        next if $used->[$i];        # if used, skip to next letter
        $out->[$recursLev] = $in->[$i];# put this letter in output
        $used->[$i] = 1;            # mark this letter as used
        DoPermute($in, $out, $used, $recursLev + 1);
        $used->[$i] = 0;            # unmark this letter
    }
}
```

从结构上讲，这个函数使用了一个名为Permute的包装函数，它分配了三个数组并对输入字符串进行了一些处理。然后，它调用递归函数DoPermute来进行递归。如果你对Perl的语法不太熟悉，那么下面这些解释应该对你有所帮助。我们声明了一个局部变量my（注意：Perl语言有一个保留字“local”，它的作用是为一个全局变量复制一个局部副本。简单地说，你只要记住“my声明了一个局部变量，而保留字“local”在没有好的理由时最好别用”就行了）。例程的参数被传递到了数组“@_”里。Permute函数的第二行声明了三个局部变量$in、$out和$used，它们将用来保存数组的读写位置（相当于数组的下标指针）。“@”是一个数组操作符，在枚举性上下文（比如if语句的条件子句）里，它将把数组的名字解释为该数组中的元素个数；在列表性上下文（比如print语句）里，它将把数组各元素的值合并在一起。

当然，这个算法用C语言也很容易实现，如下所示。把这个算法的C语言实现版本与Perl语言实现版本进行对照将进一步加深你的理解：

```
int Permute( char inString[]) {
    int length, i, *used;
    char *out;

    length = strlen(inString);
    out = (char *) malloc(length+1);
    if (!out)
        return 0; /* Failed */

    /* so printf doesn't run past the end of the buffer */
    out[length] = '\0';
    used = (int *) malloc(sizeof(int) * length);
    if (!used)
        return 0; /* Failed */

    /* start with no letters used, so zero array */
    for (i = 0; i < length; i++) {
        used[i] = 0;
    }

    DoPermute(inString, out, used, length, 0);

    free(out);
    free(used);
    return 1; /* Success! */
}

void DoPermute(char in[], char out[], int used[],
               int length, int recursLev)
{
    int i;

    /* Base case */
    if (recursLev == length) {
        printf("%s\n", out); /* print permutation */
        return;
    }

    /* Recursive case */
    for (i = 0; i < length; i++) {
        if (used[i])            /* if used, skip to next letter */
            continue;

        out[recursLev] = in[i]; /* put current letter in output */
        used[i] = 1;            /* mark this letter as used */
```

```
            DoPermute(in, out, used, length, recursLev + 1);
            used[i] = 0;              /* unmark this letter */
        }
    }
```

## 6.3 面试例题：字符串的全组合

- **请编写一个函数，用它把字符串中所有字符的各种组合形式全都显示出来。各种组合的长度范围是从一个字符到字符串的长度。不管排列顺序如何，只要两种组合中的字符完全一样，它们就是同一种组合。比如说，给定输入字符串“123”，则“12”和“31”是不同的组合，而“21”和“12”则是同一种组合。**

这个问题与字符串的字符全排列问题有很多类似之处。如果还没有把上一道面试例题弄明白，那你最好还是先把上一道面试例题解答出来再说。

参照上一道面试例题的解答思路，我们先以人工方式试一个例子，看看它能告诉我们些什么。因为你正试图从一个例子推导出一个算法来，所以你仍需要采用一种系统化的方法。你可以按它们各自的长度把各种组合列出来。我们将使用“wxyz”做为我们例子中的输入字符串。因为每种组合中的字符允许随意排列，所以我们将按它们出现在输入字符串里的先后顺序把各种组合写出来，这样就不至于把自己给弄糊涂了。

| | | | |
|---|---|---|---|
| w | wx | wxy | wxyz |
| x | wy | wxz | |
| y | wz | wyz | |
| z | xy | xyz | |
| | xz | | |
| | yz | | |

这些组合似乎有着一定的规律，但仍不明朗，看不出它能帮助我们找出一个算法。按输入字符串的顺序（就这个输入字符串而言，就是字母表顺序）来排列它们的做法曾经帮助我们解决了字符串的全排列问题。现在，我们再把上面这些组合按输入字符串的顺序排列一下，看能不能发现一些有帮助的东西：

| | | | |
|---|---|---|---|
| w | x | y | z |
| wx | xy | yz | |
| wxy | xyz | | |
| wxyz | xz | | |
| wxz | | | |
| wy | | | |
| wyz | | | |
| wz | | | |

好像能看出点意思了。输入字符串中的每一个字母都对应着一列组合形式，每列的第一种组合是输入字符串里的一个字符，而其余组合则是在该字符的右边“粘”上这一列右侧的各种组合而构成的。我们以“x”列为例来进行分析，这一列的单字符组合是“x”。出现“x”列右侧的各种组合是“y”、“yz”和“z”；如果你把这些组合“粘”在字符“x”的后面，就将得到“x”列中的其余组合：“xy”、“xyz”、和“xz”。你可以试试这条规则能不能产生出所有的组合形式来：以字符“z”做为最右边的那一列——即“z”列，然后向左前进；每前进一列，就先把输入字符串里的倒数下一个字符（依次为“y”、“x”、“w”）写在第一行上做为一种组合，然后再把当前列左边各列中的所有组合形式“粘”在当前字符的后面并把得到的各种组合依次写在这一列上。我们正确地生成了和上面一模一样的组合表。所以说，这是一个能够生成字符串全组合的办法，并且还是一个递归性的办法。这个方法需要把前一步生成的各种组合都记录或保存下来，所以存储空间方面的开销比较大；但它至少证明了一点：这个问题确实能够用递归性的办法来解决。请再仔细研究一下你写出来的组合表，看能不能从中发现一个更高效的递归算法。

请注意出现在各个位置上的字符。在每一列上，“w”、“x”、“y”、和“z”这四个字符都出现在第一位置，但“w”从没有出现在第二位置上。在第三位置上出现过的字符只有“y”和“z”，而在第四位置上出现过的字符只有“z”且只出现了一次（“wxyz”）。看出来了吗？好像能用遍历的办法来决定各位置上的字符呢：w-z在第一位置，x-z在第二位置，y-z在第三位置，z在第四位置。用例子来检验一下这个思路，看它能不能生成全部的组合形式：它成功地生成了第一列上的全部组合；但当你选择“x”做为第一位置上的字符时，这个候选算法却会从“x”开始安排第二位置上的字符，结果是生成了一个错误的组合“xx”。很明显，这个算法很需要进一步细化。

要想正确地生成“x”列上的其余组合，你必须从“y”而非“x”开始安排第二位置上的字符。当你生成“y”列（即以字符“y”做为第一位置字符的第三列）时，你必须从“z”开始安排第二位置上的字符——因为“yy”是一个非法组合，而“yx”和“yw”又与早已生成的“xy”和“wy”是同样的组合。也就是说，在为当前位置安排字符的时候，你只须从输入字符串里的、出现在当前列的当前字符后面的那个字符开始进行遍历——我们把这个字符称为“遍历起点字符”。

我们对以上这些研究做一下总结：你需要同时跟踪输出位置和输入起点位置。从以第一位置做为输出位置、以输入字符串的第一个字符做为输入起点位置开始，对任一给定位置，依次选择从输入起点位置到输入字符串最后一个字符之间的各个字符。每选定一个字符，就打印出这个组合，再通过全组合生成函数的递归调用去生成以这个组合打头的其他所有组合；新调用中的输入起点位置将被设置为当前字符的下一个字符，而输出位置则被设置为下一个位置。你应该找些例子来验证这一

系列动作能正确无误地完成任务。很好，没有问题——刚才提到的第二列问题（会生成“xx”组合的问题）已经被解决了。在动手编写代码之前，把这个算法写成一份提纲将对你有很大的帮助作用。另外，我们还需要在“断臂递归”的高效率和普通递归的良好程序设计风格之间做出选择（请参阅面试例题2“字符串的全排列”中的讨论）。对全组合算法来说，两种候选方案在性能与风格方面造成的高下差异并不像它们在全排列算法中那么显著。

```
对从输入起点位置到输入字符串最后一个字符之间的各个字符，循环
    为输出字符串的当前位置选择一个字符
    把这些字符输出到输出字符串里
    如果当前字符不是输入字符串中的最后一个字符
从下一个位置开始生成全组合，循环将从当前字符后面的下一个字符开始
```

在经过如此复杂的分析之后，我们最终得到的算法却是如此的简单！你可以开始编写代码了。如果你使用的是C语言，最好先用一个短小的包装函数来完成为输出各种组合形式而分配一个尺寸适当的输出缓冲区的工作。此外，因为你将通过这个输出缓冲区来输出一些不同长度的字符串，所以千万不要忘记在它的适当位置写上一个NUL字符（'\0'）来结束每一个字符串，这样，printf语句就不会把输出缓冲区里的“垃圾”也输出出来了。下面就是完成字符串全组合任务的代码：

```
int Combine(char inString[])
{
    int length;
    char *out;
    length = strlen(inString);
    /* allocate output buffer */
    out = (char *) malloc(length + 1);
    if (!out)
        return 0; /* failed */

    /* enter recursive portion */
    DoCombine(inString, out, length, 0, 0);
    free(out);
    return 1; /* success! */
}

void DoCombine(char in[], char out[], int length,
               int recursLev, int start)
{
    int i;
    for (i = start; i < length; i++) {
        out[recursLev] = in[i];    /* select current letter */
        out[recursLev + 1] = '\0'; /* tack on NUL for printf */
        printf("%s\n", out);
```

```
        if (i < length - 1) /* recurse if more letters in input */
            DoCombine(in, out, length, recurseLev + 1, i + 1);
    }
}
```

这个解决方案应该能让大多数面试考官满意了。不过，你还可以对DoCombine函数再进行一下优化，消除其中的if语句。因为这是一个递归算法，所以由此而得到的性能改进与函数调用的开销相比是微不足道的，但你不妨试试，看自己能不能想到并实现这一点。

类似于我们在解决全排列问题时的做法，我们把用Perl语言实现的这个函数也列在这里供大家学习和对照。全排列问题是用数组来解决的，但全组合问题我们打算直接用字符串来解决。我们用了一个针对Perl的策略：在每一次递归调用中，直接把输入字符串的剩余部分而不是它的起点位置偏移值传递给函数做为输入参数。最后，我们还把刚才提到的优化措施也实现了出来。

```
sub Combine {
    my ($in, $comb) = @_;     # Get arguments
    my ($letter, $i);         # declare local variables
    for ($i = 0; $i < length($in) - 1; $i++) {
        $letter = substr($in, $i, 1);  # Select current letter
        print $comb, $letter, "\n";
        Combine(substr($in, $i + 1), $comb . $letter);
    }
    print $comb, substr($in, length($in) - 1, 1), "\n";
}
```

请注意我们消除那条if语句的做法：去掉for循环里的最后一次递归，把有关代码挪到for循环全部结束之后。这种优化技巧总称为“循环分区”（loop partitioning），能够用循环分区技巧消除的if语句叫做“依赖于循环计数变量的条件子句”（loop index dependent conditional）。再说明一下，这种优化技巧并不能使递归算法的性能得到很大的提高，但对层层嵌套着的循环结构却十分重要。

## 6.4 面试例题：电话键单词

**•很多人在把自己的电话号码告诉别人的时候，经常会用一个单词来代替那七位电话号码数字。比如说，假设我的电话号码是866-2665，我就会告诉别人说我的电话号码是“TOOCOOL”，而不是难以记忆的7个数字。请注意，能用来代表866-2665的单词有很多种可能性（但大多数都没有什么意义）。你可以在电话的拨号盘上看到与各数字键对应的字母，如图6-2所示。**

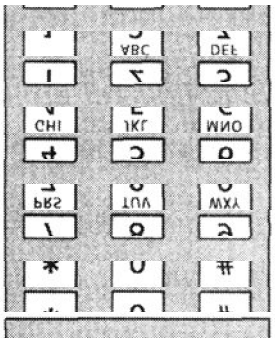


图6-2 电话上的按键

**请编写一个函数，它以一个7位数的电话号码为输入，把各种可能的“单词”——也就是能够用来代表给定号码的字母组合——都打印出来。因为电话上的“0”、“1”按键上没有字母，所以你只需把数字2-9转换为字母。**

**这个函数的调用模型必须是下面这个样子：**

```
void PrintTelephoneWords (int phoneNumber[] );
```

**其中，phoneNumber是一个有7个元素的整数数组，每个元素是电话号码中的一个数字。你可以认为传递给你函数的输入参数都是合法的电话号码。**

**你可以使用下面这个函数：**

```
char GetCharKey (int telephoneKey, int place)
```

**它以一个电话按键数字（0-9）和一个位置序号（1、2、或3）为输入参数，返回该数字按键上与指定序号相对应的字母。比如说，GetCharKey (3, 2)将返回字母“E”——电话按键“3”上的字母是“DEF”，而“E”正好是这几个字母当中的第二个。**

为了便于讨论，我们先定义几个词汇。大家知道，电话号码是由数字组成的，每三个字母对应于一个数字（按键“0”和“1”上没有字母，它们将直接出现在电话键单词里，所以我们也把它们叫做字母）。我们就将把电话按键上的第一、第二、第三个字母分别称为该按键数字的低值、中值、和高值。你要做的事情是按照给定的电话号码“创造”单词——也就是由字母组成的字符串。

首先，用你的数学知识“吓唬”一下面试考官——告诉他与7位电话号码对应的单词应该是多少个。这是一个与排列组合有关的算术题；可万一你把有关公式忘得差不多了，就不要直接去冒这个险——先自己归纳一下。先来看只有一个数字的电话号码；很明显，将会有三个单词。再来试试有两个数字的电话号码，比如说“56”：第一个字母有三种可能性，第二个字母也有三种可能性，它们的组合是9种；也就是说，总共有9个单词与这个号码相对应。于是，电话号码每增加一位数字，与之对应大苏打单词总数就要增加3倍。因此，7位电话号码将对应于$3^7$个单词；$n$位电话号码将对应于$3^n$个单词。因为按键“0”和“1”上没有字母，所以包含有“0”和“1”的电话号码所对应的单词要少一些，但$3^7$将是7位电话号码所对应的单词总数的上限。

接下来，你需要想出一个算法来把这么多的单词打印出来。大家不妨先随便找个电话号码（比如我们上大学时的电话号码497-1927），并为它“创造”几个单词试试。最容易想到的办法是按字母顺序来排列这些单词。这个办法的好处是你知道下一个单词是什么，因而不太可能会有所遗漏。你已经知道这个电话号码有$3^7$个对应的单词，所以你肯定没有时间把它们全都写出来；那就把按字母顺序最开始的那几个和最末尾的那几个写出来好了。电话键单词的第一个字母应该从电话键上的第一个字母开始，这将保证你写出来的第一个单词也就是按字母表顺序排列出来的第一个单词。因此，对应于号码“497-1927”的第一个单词将是“GWP1WAP”；其中，

“G”是按键“4”上的字母“GHI”中的第一个，“W” 是按键“9”上的字母“WXY”中的第一个，“P”是按键“7”上的字母“PRS”中的第一个……依次类推。

如此不停地写下去，你最终将得到一个下面这样的列表：

GWP1WAP
GWP1WAR
GWP1WAS
GWP1WBP
GWP1WBR
…
IYS1YCR
IYS1YCS

这份清单很容易生成，这道试题的解决方案似乎也不难确定。但要想把算法转变为代码却不那么容易。我们就从上面这份清单开始来研究一下前一个单词是怎样转变为后一个单词的吧。

在这份按字母表顺序排列的清单中，你知道它的第一个单词是什么，如果再把由它到第二个单词的转变规律找出来，你就能用这个算法把使用的单词都生成出来。对比前、后两个单词就会发现，最后一个字母一直在变化——不停地按P-R-S的次序做循环。当最后一个字母从“S”循环回“P”时，它左边的那个字母就会发生变化。再仔细研究一下这个特点，看能不能由此总结出一条普遍的规律来。用一些例子来做分析是个好办法，也许还需要得写出更多的单词才能看出其中的规律来（三位数的电话号码应该够用了，用上面这份清单当然更没问题）。我们首先发现了这样一个规律：当某位置上的字母发生变化后，它右面的字母将轮转遍它的可取值，然后这个字母才会再次变化；反过来说，当某位置上的字母变化为它的最低值时，它左边的字母将变化为下一个值。

根据上述分析，有两条路可以让你找到解决这个问题的算法。你可以从第一个字母开始，然后让某个字母影响它右边的字母；或者，你可以从最后一个字母开始，让某个字符影响它左边的字母。两条路都很合理，但你必须从中挑选一个。我们决定选择第一条路，看它能把我们带到哪儿去。

现在，你必须好好想想自己到底想干些什么。你手里有一个经观察得到的结果：当某位置上的字母发生变化时，它右面的字母将轮转遍它的可取值，然后这个字母才会再次变化。现在，我们从这个观察结果出发，看怎样才能从按字母表顺序排列的前一个单词到达下一个单词。我们现在把观察到的结果总结一下：当位置$i$处的字母发生变化时，位置$i+1$处的字母将轮转遍它的可取值。当某个算法需要用位置$i$和位置$i+1$之间的关系来描述时，就表明了一种递归的迹象，所以我们现在应该朝着递归

算法的方向去努力。

其实，我们已经把这个算法的大部分都分析出来了。我们知道，某位置上的字母将影响到它右边的字母，只要再把如何开始这一过程以及递归算法的基底情况找出来就完整了。我们还是得从单词清单入手来研究如何开始这一处理过程。首先，第一个字母只循环一次。因此，如果你从第一个字母开始循环，就会引起第二个字母的多次循环，进而引起第三个字母的更多次循环。当你改变了最后一个字母时，就没有可循环东西了，所以这应该是一种能够结束递归的基底情况。在进入基底情况后，你还应该把当前的单词打印出来——因为你已经把字母表顺序中的下一个单词生成出来了。另外，别忘了对特例——即给定的电话号码里有数字“0”和“1”时的情况——进行处理：同一个单词不应该被输出三次，所以你必须对这一特例进行检查，并在遇到它的时候立即前进到下一个字母的循环上。

根据以上分析，我们得到了一个下面这样的算法：

```
如果当前数字超出了最后一位数字
   输出当前单词，因为你已经到达了整个算法的末尾
否则
   对当前数字所对应的三个字母，按从低到高的顺序循环
       用字母代替当前数字
       前进到下一个数字并递归
       如果当前数字是“0”或“1”，返回
```

实现这个算法的代码如下所示：

```
#define PHONE_NUMBER_LENGTH 7

void PrintTelephoneWords(int phoneNum[])
{
    char result[PHONE_NUMBER_LENGTH + 1];

    /* tack on the NUL character at the end */
    result[PHONE_NUMBER_LENGTH] = '\0';

    DoPrintTelephoneWords(phoneNum, 0, result);
}

void DoPrintTelephoneWords(int phoneNum[], int curDigit,
    char result[])
{
    int i;

    if (curDigit == PHONE_NUMBER_LENGTH) {
        printf("%s\n", result);
        return;
```

```
    }

    for (i = 1; i <= 3; i++) {
        result[curDigit] = GetCharKey(phoneNum[curDigit], i);
        DoPrintTelephoneWords(phoneNum, curDigit + 1, result);
        if (phoneNum[curDigit] == 0 ||
                phoneNum[curDigit] == 1) return;

    }
}
```

这个算法的执行时间如何？它不可能小于$O(3^n)$，因为有$3^n$种单词组合；所以任何一个正确的解决方案至少是$O(3^n)$级的。生成每个新单词所需要的时间是固定的，所以这个算法的运行时间的确是$O(3^n)$。

**• 重新实现PrintTelephoneWords函数，不允许使用递归。**

刚才的递归算法好象帮不上这里的忙。递归函数来源于你归纳出来的递归算法，递归算法又表现为你写出来的算法步骤。当然，你可以人为地采用一种基于堆栈的数据结构来模拟递归函数的工作流程，但另外还有一种更好的算法在等你去发现也说不定。在递归方案里，我们是按由左向右的顺序来解决这道面试题的。我们的观察告诉我们，还存在着一种按由右向左的顺序来解决这一问题的办法。经观察，当某位置上的字母从高值变成低值时，它左边的字母就会递增一次。这一观察结果能否帮助我们为这个问题找出一个非递归的算法呢？

我们仍要用一些例子来找出如何生成字母表顺序中的下一个单词的办法。既然是从右向左进行，所以我们必须密切注意单词右边发生的事情，看它在算法生成下一个单词时有什么变化规律。根据前面的观察，最右边的字母总是在不停地变化着。这样看来，依次切换最后一个字母似乎是一个好的开端。当最后一个字母已经是它的最高值而你又对它进行递增时，它将返回自己的最低值，它左边的字母（即倒数第二个字母）将递增。那么，如果倒数第二个字母也已经是它的最高值时会怎样呢？根据前面的单词清单，这个字母也将返回它的最低值，它左边的字母（即倒数第三个字母）将递增。这一过程不断重复，直到没有字母需要返回它自己的最低值位置。

这正是我们要找的算法，但还必须把算法的开始和结束部分的细节弄清楚才行。算法的开始部分比较好办，以手动方式构造出第一个单词——就像你列出那份单词清单时一样——不就行了吗。结束部分怎么办呢？请仔细研究一下清单里的最后一个单词，当你试图递增它的时候，会发生什么事情？说对了，每个字母都将被重置为它的最低值。你可以检查每一个字母是否等于它们各自的最低值，但这样做好像效率很低的样子。仔细研究一下就会发现，到你把所有单词全都生成并打印出来的时候，第一个字母只被重置了一次；你可以把这一点当做你已完成全部工作的信号。注意，千万不要忽略第一个字母也可能是数字“0”或“1”的情况。“0”和“1”是无法递增的

（它们总是“0”或“1”），所以应该把“0”和“1”一直当做它的最高字母值来对待而直接去递增它左边的字母。根据以上分析，我们得到了一个下面这样的算法：

```
一个字母一个字母地“创造”出第一个单词
无限循环
    输出当前单词
    递增最后一个字母，并对它左边的字母做相应的改变
    如果第一个字母已经重置，结束循环——你已完成了这一任务
```

下面是这个算法的代码实现。（如果你把每个数字按键的低、中、高值保存在变量里，就用不着调用GetCharKey函数去检查它的值是低、中、还是高了，这将减少一些函数调用方面的开销。这将给算法稍微增加一些难度，而且，就这个算法而言，这类优化的效果是微乎其微的——别忘了，这是一个$O(3^n)$级的算法。）

```
#define PHONE_NUMBER_LENGTH 7

void PrintTelephoneWords(int phoneNum[])
{
    char result[PHONE_NUMBER_LENGTH + 1];
    int i;

    /* Initialize the result (in our example,
     * put GWP1WAR in result).
     */
    for (i = 0; i < PHONE_NUMBER_LENGTH; i++)
        result[i] = GetCharKey(phoneNum[i], 1);

    /* Tack on the NUL character at the end. */
    result[PHONE_NUMBER_LENGTH] = '\0';

    /* Main loop begins */
    while (1) {
        printf("%s\n", result);

        /* Start at the end and try to increment from right
         * to left.
         */
        for (i = PHONE_NUMBER_LENGTH - 1; i >= -1; i--){
            /* You're done because you
             * tried to carry the leftmost digit.
             */
            if (i == -1) return;

            /* Otherwise, we're not done and must continue. */

            /* You want to start with this condition so that you can
```

```
                * deal with the special cases, 0 and 1, right away.
                */
               if (GetCharKey(phoneNum[i], 3) == result[i] ||
                   phoneNum[i] == 0 || phoneNum[i] == 1) {
                   result[i] = GetCharKey(phoneNum[i], 1);
                   /* No break, so loop continues to next digit */
               } else if (GetCharKey(phoneNum[i], 1) == result[i]){
                   result[i] = GetCharKey(phoneNum[i], 2);
                   break;
               } else if (GetCharKey(phoneNum[i], 2) == result[i]){
                   result[i] = GetCharKey(phoneNum[i], 3);
                   break;
               }
           }
       }
   }
```

这个算法的执行时间如何？

因为要生成$3^n$个单词，所以只要算法是正确的，它的执行时间就不可能低于$O(3^n)$。生成每个单词的开销是一定的；检查数字按键的低、中、高值会稍微增加开销，但也是一个常数。因此，它也是一个$O(3^n)$级的解决方案。

# 其他程序设计问题

到目前为止，我们见过的都是一些比较常见的面试问题。还有一些试题，虽说不那么常见，但在程序设计面试中也经常会遇到。因为涉及面很广，所以本章的内容也就比较松散；但我们会把同类问题集中到一个小节里进行讨论，希望能对大家有所帮助。

## 7.1 计算机图形

计算机的显示画面是由光栅点阵组成的，这些点阵构成了一个坐标系统。计算机图形算法通过改变一组点阵的颜色来变换画面，这些算法的基础是各种各样的几何公式。因为计算机屏幕上的点阵个数是有限的，所以把几何公式转换为点阵图形的工作相当复杂。几何公式中的数值大都是实数（浮点数），而点阵却是以固定间隔分布在显示器屏幕上的，所以几何公式的计算结果必须被调整为点阵坐标——也就是把浮点数舍入为整数。但把浮点数舍入为最接近的点阵坐标往往并非最佳方案。计算机图形算法中的舍入方案大都不同寻常，往往要增加一些调整因子。如果舍入方案考虑不周，就会把本应连续的线条显示为断续的线段。在实现这类算法的时候，一定要慎重选择你的舍入方案和调整措施，避免图形出现扭曲或者断续。

就拿画直线这个最简单的任务来说吧。假设面试考官让你编写一个用来在两点之间画出一条直线的函数。在做了一些研究和分析之后，你轻而易举地得到了一个几何公式 $y = mx + b$，然后根据$x$坐标的取值范围计算出了$y$坐标的值并画出了构成这条直线的点。你也许会想：这个函数太容易写了；可真的是这样吗？

陷阱就隐藏在这个问题的细节里。首先，你必须考虑到垂直线的问题。此时，$m$等于无穷大，你的函数能画出这条直线吗？类似的情况还有：直线并不垂直，但却非常接近于垂直——比如说，直线的横坐标区间只有两个点阵，可纵坐标区间却有20

个点阵。如果你只能画出两个点，那它还能叫做直线吗？为了解决这类问题，你决定再写一个例程：先把直线公式变换为 $x = (y - b) / m$；然后，当需要画一条接近于垂直的直线时，就用新例程以y坐标来计算x坐标，当需要画一条接近于水平的直线时，就仍使用旧例程。

可即便如此也不能解决所有的问题。假设你要画一条斜率为1的直线，即$y = x$。此时，不管使用哪个例程，你都会得到（0, 0）、（1, 1）、（2, 2）……等点阵坐标。这在数学上是正确的，但在屏幕上画出来的直线却不太好看——与其他直线相比，这条直线的点阵分布得过于稀疏：一条长度为100的对角线里的点阵比一条长度为80的水平线里的点阵还要少。而一个优秀的画直线算法至少应该保证画出来的直线都有差不多的点阵密度才行。另外一个问题与浮点数的舍入有关。如果你计算出来的坐标是（0.99, 0.99）并使用了一个类型转换（type cast）来把它转换为整数，这一点就将被画在屏幕坐标的（0, 0）位置上。因此，你必须采用其他的舍入方案，好让这一点能被画在（1, 1）位置上。

与图形有关的问题有着数不胜数的特例；只有明白了这一点，你才算是真的明白了。做为画直线问题的结论，我们想告诉大家这样一句话：即使你把我们刚才提到的各种特例都考虑到了，你的画直线算法也不能算做是最好的。在编写与计算机图形有关的程序时，除了会遇到刚才提到的各种困难和特例之外，你还必须考虑到浮点运算的一个特点——慢。如果全部使用整数进行计算，算法的性能当然会大幅提高，但它的复杂性也将比我们这里讨论的情况高出一大截。

**提示：** 计算机图形涉及到几何公式的计算结果与屏幕坐标之间的转换问题。千万不要忘记检查舍入错误、线条断续以及各种特例情况。

## 7.2 位操作符

很多计算机语言都允许程序员直接访问变量中的某个位，并为此提供了相应的手段。在日常的程序设计工作中，需要用到位操作符的场合并不很多；但在程序设计面试中，它们却相当常见。总之，它们很值得复习。

不同的程序设计语言有不同的位操作符。因为C语言是主流程序设计语言中最底层的，所以C程序里的位操作相对来说也最为多见。因此，我们的复习就将以C语言中的位操作符为重点。C语言中的位操作符与C++中的完全一致，与Java中的位操作符也基本一致（惟一的区别是：在Java里，“>>”操作符总是做符号位扩展，并且增加了一个用来进行0扩展右移位的“>>>”操作符；在C语言里，“>>”操作符的行为要取决于它的操作数是否属于带符号类型。）

位操作符的处理对象是二进制位，所以你的思路就必须从位入手。在计算机即，

数值通常被表示为二进制补码。与二进制打交道的人对补码这种记号方法是不应该感到陌生的。二进制补码与普通的二进制数并没有什么区别，事实上，用这两种记号方法表示出来的正整数是一模一样的。它们之间的区别表现在负数的表示方法上。现时期，整数基本上都是32位的，但为了便于讨论，我们将以8位的整数做为例子。在二进制补码记号中，正整数13将被表示为00001101，与正常的二进制记号完全一样；负数的表示方法就比较奇怪了。当需要表示一个负数的时候，它的补码将是“翻转该数字正整数记号中全部的位，然后再加上1”而得到的记号。就拿－1来说吧，它的正整数记号是00000001，翻转其中全部的位后将得到11111110，再给它加上一个1就得到了－1的补码11111111。不熟悉二进制补码的人往往会对这种做法感到奇怪，可这样做是有很多好处的——尤其是在做减法的时候。请注意，第一个位是一个符号位：如果它是0，就表明这是一个正数；如果它是1，就表明它是一个负数。

位操作符将直接对变量的位进行操作。对常见的位操作符是一元操作符“~”，即NOT（非）操作符。这个操作符的作用是对其操作数的所有位进行取反。也就是说，每个1都将变成0，而每个0又会变成1。比如说，如果让“~”操作符作用于00001101，那么结果将是11110010。

另外三个位操作符是“|”（或，OR）、“&”（与，AND）、和“^”（异或，XOR）。它们是一些逐位操作的二元操作符，也就是说，操作结果中的第*i*个位将同时取决于前一个数的第*i*个位和后一个数的第*i*个位。这些操作符的运算规则如下所示：

“&”：如果两个位都是1，则结果为1；否则，结果为0。比如说：

```
  01100110
& 11110100
  01100100
```

“|”：只要有一个位是1，结果就将是1；如果两个位都是0，结果就将是0。比如说：

```
  01100110
| 11110100
  11110110
```

“^”：如果两个位相同，则结果为0；如果两个位不同，则结果为1。比如说：

```
  01100110
^ 11110100
  10010010
```

剩下来的两个位操作符是“>>”（右移位）和“<<”（左移位），它们的作用是把操作数中的位分别向右或向左移动给定的次数。在使用“<<”操作符的时候，因移动位而产生的空缺将用0来填补。在使用“>>”操作符的时候，如果操作数是一个无符号数值，则用0来填补空缺；如果操作数是一个带符号数值，则用符号位来填补空

缺。换句话说，如果操作数是一个正整数，那么空缺将用0来填补；如果操作数是一个负整数，那么空缺将用1来填补。比如说，“01100110 << 5”的操作结果是“11000000”；“01100110 >> 5”的操作结果是“00000011”；但负数“10100110 >> 5”的操作结果却是“11111101”——请注意符号位的扩展情况。

利用移位操作符能够非常迅速地完成乘以2或除以2的计算：右移一个位相当于除以2；左移一个位相当于乘以2。移位操作与乘除操作的对应关系对十进制数字也同样适用。请看十进制数字17的例子：“17 << 1”的结果是170，与“17 × 10”的效果完全一样；而“17 >> 1”的结果是1，与用10对17做整数除法的效果也完全一样。

## 7.3 结构化查询语言

结构化查询语言（Structured Query Language，SQL）是用于关系数据库的处理语言，几乎能够完成各种类型的数据库操作。SQL是一个范围广阔的话题，光是传授别人如何使用SQL语言的书籍就多如牛毛。不过，用SQL来完成基本的数据存储和检索操作却并不复杂。

如果你在自己的简历或者面试过程中没有说起过自己懂得SQL，那么面试考官通常就不会向你提出有关SQL的问题。这本书并不教你如何使用SQL，我们只是想帮大家复习一下SQL的重要概念而已。我们的讨论偏重于常见的面试内容和例题，对SQL语法定义方面的事情不细讲。

**提示：** 如果你没有说起过自己懂得SQL，面试考官通常就不会向你提出有关SQL的问题。

与数据库有关的面试题通常是让你写出一些查询来检索数据库中的某些数据。数据库的工作区（sechema）结构通常会在已知条件中给出，所以一般不会让你去设计一个工作区的结构。在这一小节里，我们给出的例子都将在如下所示的工作区里进行操作：

```
Player(name CHAR(20), number INT(4));
Stats(number INT(4), totalPoints INT(4), year CHAR(20));
```

表Player（意思是“运动员”）里的样本数据如表7-1所示，表Stats（意思是“统计”）里的样本数据如表7-2所示。

**表7-1　Player表中的样本数据**

| NAME（姓名） | NUMBER（号码） |
|---|---|
| Larry Smith | 23 |
| David Grozalez | 12 |
| Geroge Rogers | 7 |
| Mike Lee | 14 |
| Rajiv Williams | 55 |

表7-2 Stats表中的样本数据

| NUMBER（号码） | TOTALPOINTS（总得分） | YEAR（年级） |
|---|---|---|
| 7 | 59 | Freshman |
| 55 | 90 | Senior |
| 22 | 15 | Senior |
| 86 | 221 | Junior |
| 36 | 84 | Sophomore |

“INSERT”是最基本的SQL语句之一，它的作用是把一个数据项插入到表里去。比如说，如果你想把一位姓名是“Bill Henry”、号码是“50”的运动员插入到Player表里去，就需要使用如下所示的语句：

```
INSERT INTO Player VALUES('Bill Henry', 50);
```

“SELECT”是程序设计面试中最常用到的SQL语句，它的作用是把有关数据从表里检索出来。比如说，下面这条语句：

```
SELECT * FROM Player ;
```

将把Player表里的全部数据都检索出来，如下所示：

```
+-------------------------+
| name           | number |
+-------------------------+
| Larry Smith    |     23 |
| David Gonzalez |     12 |
| George Rogers  |      7 |
| Mike Lee       |     14 |
| Rajiv Williams |     55 |
| Bill Henry     |     50 |
+-------------------------+
```

下面这条语句将把你指定的列检索出来：

```
SELECT name FROM Player;
```

检索结果如下所示：

```
+----------------+
| name           |
+----------------+
| Larry Smith    |
| David Gonzalez |
| George Rogers  |
| Mike Lee       |
| Rajiv Williams |
| Bill Henry     |
+----------------+
```

你也许会遇到让你把某些特定的数据检索出来的情况。比如说，如果面试考官让你把号码小于10或大于40的运动员的名字查出来，你就要使用下面这条语句：

```
SELECT name FROM Player WHERE number < 10 OR number > 40;
```

检索结果如下所示：

```
+----------------+
| name           |
+----------------+
| George Rogers  |
| Rajiv Williams |
| Bill Henry     |
+----------------+
```

此外，你经常会遇到需要对两个甚至多个表里的数据进行检索的情况。比如说，如果面试考官让你把全体运动员的姓名和他们各自的总得分检索出来，你就需要通过number字段把Player和Stats表结合起来进行检索。人们把这种情况中的number字段称为“共用键”（common key），因为它是这两个表里既完全一致又独一无二的值。下面就是能够完成这一检索任务的查询语句：

```
SELECT name, totalPoints FROM Player, Stats WHERE
Player.number = Stats.number;
```

检索结果如下所示：

```
+-------------------------------+
| name           | totalPoints |
+-------------------------------+
| George Rogers  |          59 |
| Rajiv Williams |          90 |
+-------------------------------+
```

统计操作——例如MAX、MIN、SUM、AVG等——是另外一些比较常用的SQL功能。这几个统计功能的作用分别是找出或者计算出某数据列中的最大值、最小值、总和、平均值。比如说，如果面试考官让你把每位运动员的平均得分统计出来，你就需要使用下面这样的查询：

```
SELECT AVG(totalPoints) FROM Stats;
```

统计结果如下所示：

```
+------------------+
| AVG(totalPoints) |
+------------------+
|          93.8000 |
+------------------+
```

你也许会遇到让你对某个数据子集进行统计的情况。比如说，如果面试考官让

你把运动员的平均得分按年级统计出来，你就要用到“GROUP BY”子句，就像下面这个查询那样：

```
SELECT year, AVG(totalPoints) FROM Stats GROUP BY year;
```

统计结果如下所示：

```
+-----------------------------+
| year       | AVG(totalPoints) |
+-----------------------------+
| Freshman   |          59.0000 |
| Junior     |         221.0000 |
| Senior     |          52.5000 |
| Sophomore  |          84.0000 |
+-----------------------------+
```

一般来说，与SQL有关的程序设计面试题主要集中在前面给出的各种插入和查询语句上，涉及到“UPDATE”语句、“DELETE”语句、访问权限、数据安全、数据库设计、并发处理或者查询优化等方面的SQL面试题是比较少见的。

## 7.4 并发程序设计技术

并发程序设计技术是另一个“如果你不说自己有这方面的经验，一般就不会被问到”的话题，同时也是一个庞大而又困难的话题，所以我们也将只涉及它的重点概念，为那些比较熟悉这一领域的人们提供一个快速复习的机会。

如果你没有说起过自己有设计并发程序的经验，面试考官通常就不会向你提出这方面的问题。

并发是一项非常有用的技术，它允许多个线程（thread）共享中央处理器资源和程序中的变量。举个例子：假设你想用一台计算机完成两项任务。任务A需要等待用户提供输入数据，而任务B则用不着用户提供输入数据但要做大量的运算。如果计算机是依次去执行这两个任务——即先执行完其中的某个之后再去执行另一个的话，任务A就会在计算机等待输入数据时浪费掉大量的处理器时间。线程技术允许计算机利用任务A中的等待时间去执行任务B，所以线程化的任务要比依次执行的任务能更为有效地共享计算机资源。

我们用一个银行系统为例来演示一下这个概念。这个系统由一个运行在一台中央计算机上的程序构成，这个程序控制着分布在多个地点的多个柜员机。每个柜员机都有它自己的线程，银行的出纳员能同时使用各自的柜员机来共享该银行的账户数据。

在这类场合里，如果线程的控制机制设计不当，就会导致很多问题。比如说，我们假设某银行的柜员机程序里有一个从储户的账户里扣除他取款数额的函数，而

储户的存款余额保存在全局变量userBalance里。

```
int Deduct(double amount)
{
    double newBalance;
    if (amount < userBalance) {
        return 0; /* Insufficient funds */
    } else {
        newBalance = userBalance - amount;
        userBalance = newBalance;
        return 1;
    }
}
```

现在，假设有一对夫妻——丈夫叫Ron，妻子叫Sue——的开户银行使用了个函数，于是就发生了下面这样的故事：起初，Ron和Sue的账户里有$500美元。他们两人分别去到位于不同地点的银行分行，并打算每人提取$100美元。首先，Ron告诉银行出纳员说他想提取$100美元，于是出纳员进行了从Ron和Sue的账户里扣除$100美元的操作，但这个线程在银行的中央计算机执行完下面这条语句后被切换出去了：

```
newBalance = userBalance - amount;
```

此后，中央计算机切换到了Sue那里，而她也正在提取$100美元。当负责接待她的出纳员扣除$100美元的时候，账户余额仍然是$500美元——因为变量userBalance的值此时还没有被更改过。Sue的线程一直执行到这个函数的末尾，并把userBalance的值更改为$400美元。现在，中央计算机又切换回Ron的交易。在Ron的线程里，变量newBalance的值依然是$400美元，于是Ron的线程就在把这个值赋给变量userBalance之后返回了。现在，Ron和Sue总共从账户里提取了$200美元，可他们的账户余额却是$400美元，也就是说，中央计算机只记录下一次$100美元的提款操作。这对Ron和Sue来说是件好事，但对银行来说却是一个非常严重的问题。

要想避免类似的情况，就必须对程序关键代码的访问情况和共享资源的使用情况加以控制和保护。C语言解决这一问题的常用办法是用一个具备有信号量API的线程代码包把关键代码部分给“包“起来。这类代码包至少要提供三个主要方法：一个用来告诉线程说必须等待某信号量（semaphore）的函数，通常就叫做wait（意思是“等待”）；一个用来触发信号量的函数，通常就叫做signal（意思是“发信号”）；一个用来创建一个新信号量的函数，通常就叫做newSemaphore（意思是“新信号量”）。

newSemaphore函数必须把它创建出来的信号量初始化为一个给定的值，这个值对该信号量上的线程个数做出了规定。

当线程在某个信号量上调用了wait函数时，如果这个信号量的值小于或等于0，这个线程就必须进入休眠等待状态，直到这个信号量变成大于0为止。一旦信号量的

值变成了大于0，等待这一信号量的某个线程就会“苏醒”。苏醒后的线程将先对这个信号量做减1操作，然后进入受保护的关键代码部分。

在这个与银行有关的例子里，应该每次只允许有一个线程去访问全局变量userBalance。添加信号量以后，我们将得到一个如下所示的函数：

```
void init()
{
    balanceLock = newSemaphore(1);
}

int Deduct(double amount)
{
    double newBalance;
    wait(balanceLock);
    if (amount < userBalance) {
        signal(balanceLock);
        return 0; /* Insufficient funds */
    } else {
        newBalance = userBalance - amount;
        userBalance = newBalance;
        signal(balanceLock);
        return 1;
    }
}
```

在编写多线程程序的时候，要特别注意“死锁”（deadlock）问题。死锁是指这样一种情况：某信号量上的线程全都处于休眠等待状态，没有一个线程能够苏醒并继续执行。比如说，如果信号量balanceLock被错误地初始化为0而不是1，就会导致死锁现象，Deduct函数中的线程都将卡在这个信号量上，谁都无法执行到底。

不同的程序设计语言在实现并发功能方面有不同的特点。比如说，Java语言中的并发功能是通过保留字“synchronized”实现的，这个保留字一般被添加在方法（method，即绑定在类或对象上的函数）的定义上。这将确保任何一个调用了这个方法的线程能够在其他线程进入这个对象（如果这是一个绑定在实例上的方法）或者这个类（如果这是一个绑定在类上的方法）上的某个“synchronized”方法之前能够执行完毕。从原理上讲，保留字“synchronized”等于是给对象或者类加上了一个二元锁（binary lock）。在设计Java程序的时候，经常会遇到需要让一个线程等待某个特定条件变化为真的情况；而这需要通过调用wait方法来实现。这个方法的作用是让这个线程进入休眠等待状态并在未来的某个时刻苏醒过来；等到达这一时刻时，你可以用notify或notifyAll方法来唤醒休眠中的线程。

## 7.5 面试例题：绘制八分之一圆形

**• 请编写一个函数来绘制八分之一圆形。这个圆的圆心位于坐标原点（0，0），**

**它的半径由输入参数给定，而所谓的“八分之一”则是指从钟表的12:00点到1:30分之间的弧形区域。这个函数的调用模型必须是如下所示的样子：**

```
void DrawEighthOfCircle(int radius);
```

**图7-1是上述坐标系统和圆弧的一个示意图。用来绘制图形显示点阵的函数调用模型如下所示：**

```
void SetPixel(int xCoord, int yCoord);
```

这道面试题的内涵比你想像的要深。如果让你去实现一个绘制完整圆形的例程，那你肯定想把计算工作压缩到最小以追求最优的效率。如果你能画出某个圆形的八分之一，就可以根据对称原理来确定这个圆形的其余弧段——如果点$(x, y)$位于圆上，那么$(-x, y)$、$(x, -y)$、$(-x, -y)$、$(y, x)$、$(-y, x)$、$(y, -x)$、$(-y, -x)$等7个点也将位于圆上。这道面试题是“扫描转换”（scan conversion）类问题中的一个颇具代表性的例子，这类问题的特点是需要把几何图形的绘制工作转换为基于点阵的对称图像。

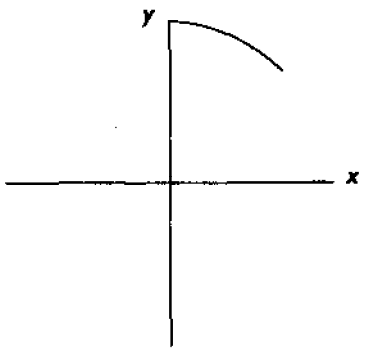



图7-1　DrawEighthOfCircle函数的输出效果

首先，你需要一个几何方程来计算圆上的点的坐标。能够产生一个圆形的常用几何方程是：

$$x^2 + y^2 = r^2$$

这个方程正是我们需要的东西——它里面的横坐标、纵坐标、和半径正好与这道面试题和给定的坐标系统相吻合。但你还得根据方程$x^2 + y^2 = r^2$推导出一个能够用来为圆形确定点坐标（$x, y$）的公式才行。要想求出一个点的坐标，最简单的办法是先设定其中的一个、再计算出另外一个来。用$y$坐标来求$x$坐标的做法不太理想；因为在经过扫描转换之后，每个$y$坐标都将产生正负两个$x$坐标，所以我们认为你应该用$x$坐标来求$y$坐标。在经过一番推导之后，我们得到了下面这个$y$坐标计算公式：

$$y = \pm\sqrt{r_2 - x^2}$$

这道面试题只需用到$y$坐标的正值，所以我们舍掉负平方根，得到：

$$y = \sqrt{r_2 - x^2}$$

如果$x$坐标是3，圆的半径是5，则 $y = \sqrt{5^2 - 3^2} = 4$。好了，用$x$坐标来计算$y$坐标的事情就解决了。接下来需要确定$x$坐标的取值范围。我们已经知道$x$坐标将从0开始，

但它会在哪儿结束呢？请仔细观察面试题所给出的示意图，你能看出怎样才能确定自己已经到达八分之一个圆的结束位置了吗？通过观察，我们发现了这样一个规律：在到达八分之一圆形的结束位置之前，*x*坐标将小于*y*坐标；在超出八分之一圆形的结束位置之后，*x*坐标将小于*y*坐标；在八分之一圆形的结束位置上，*x*坐标等于*y*坐标。因此，你可以把x坐标的取值范围确定为从0开始，到*x*>*y*时结束。把这些东西放到一起，你就能得出一个绘制八分之一圆形的算法来，如下所示：

从*x*=0，*y*=*r*开始
当*y*>*x*时，循环
　根据公式 $y=\sqrt{r_2-x^2}$ 计算出*y*坐标
　绘制点阵（*x*, *y*）
　递增*x*坐标

这个算法看起来没有什么毛病，但里面却隐藏着一个重大缺陷：这个算法把*y*坐标当做整数来对待，而实际求出来的*y*坐标却大都是一些小数。比如说，如果*y*坐标的值是9.99，那么函数SetPixel将把它截短为9而不是像你期望的那样把它舍入为10。消除这一缺陷的办法之一是设法让*y*坐标被舍入为最接近的整数，比如说，在调用函数SetPixel之前先给y坐标加上0.5。

在做出这一改动之后，我们画出来的东西将更像一个圆形。下面是实现这个算法的代码：

```
void DrawEighthOfCircle(int radius)
{
    int x, y;
    x = 0;
    y = radius;
    while (y <= x) {
        y = sqrt((radius * radius) - (x * x)) + 0.5;
        SetPixel(x, y);
        x++;
    }
}
```

这个算法的执行效率怎么样？它的执行时间是O(*n*)，*n*是需要绘制的坐标点的个数。要想正确地绘制出一个圆形，任何一种算法都至少要对函数SetPixel做*n*次调用；所以这已经是最佳的执行时间了。在while循环的每次遍历中，这个函数还用到了sqrt函数和乘法运算。sqrt函数和乘法运算都是比较慢的操作，所以这个算法可能不太适用于那些速度高于一切的图形应用领域。不需要反复调用sqrt之类的慢速函数、也不需要进行乘法运算的圆形绘制算法是存在的，这些算法的速度无疑会更快；但面试考官往往不会指望你能在程序设计面试中实现出一个这样的算法来。

## 7.6 面试例题：矩形是否重叠

- **给定两个矩形，每个矩形的位置都由它的左上角（UL）和右下角（LR）坐标来确定，且两个矩形的边都与*x*轴或*y*轴平行，如图7-2所示。请编写一个函数来判断这两个矩形是否重叠。如果两个矩形发生重叠，函数将返回1；如果不重叠，则返回0。你可以使用下面给出的结构定义：**

```
struct point {
    int x;
    int y;
};
struct rect {
    struct point UL;
    struct point LR;
};
```

请使用如下所示的函数调用模型：

```
int Overlap(struct rect A, struct rect B);
```

在动手解决这道试题之前，最好先来分析一下矩形及其相互关系的特点。首先，知道了矩形的左上角（UL）和右下角（LR）坐标后，它的右上角（UR）和左下角（LL）坐标也就不难知道了。右上角的横坐标等于右下角的*x*坐标，纵坐标等于左上角的*y*坐标；而左下角的横坐标等于左上角的*x*坐标，纵坐标等于右下角的*y*坐标。

其次，能否判断出某个点是否落在某个矩形区域内也很有用。如果一个点落在一个矩形区域内，这个点的*x*坐标就必然落在矩形左上角和右下角的*x*坐标之间，而它的*x*坐标则必然落在矩形左上角和右下角的*y*坐标之间。这一事实可以从图7-2里看出来：点“1”落在矩形A的区域内。现在，我们要开始解决这道面试题了。

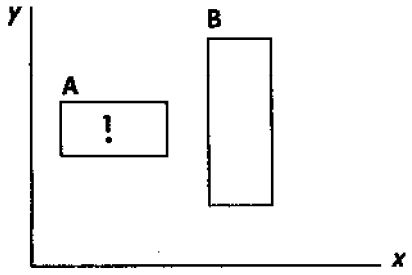


图7-2　坐标系统与矩形的示意图

这道试题看起来相当直白。我们先来看看两个矩形之间会出现哪些种重叠情况，并把它们归结为几大类。如果两个矩形彼此重叠，它们的角会呈现什么样的关系呢？这似乎是个比较不错的出发点。我们来数一下，看重叠着的矩形会有多少个角点落在对方的区域内。必须考虑的情况有以下几种：当两个矩形发生重叠时，其中的一个在对方的区域范围内有0、1、2、3、4个角。我们依次对这几种情况进行分析。首先，两个重叠着的矩形没有角点落在对方的区域范围内，如图7-3所示。

两个矩形彼此重叠、却没有任何一个角点落在对方区域范围内的情况有什么特

点呢？首先，那个比较宽的矩形肯定要比那个比较窄的矩形矮。其次，两个矩形的位置必须使它们能够发生重叠。准确地说，就是那个较窄的矩形的x坐标必定落在那个较宽的矩形的x坐标之间，而较矮的矩形的y坐标则必定落在较高的那个矩形的y坐标之间。当以上条件全部满足时，两个矩形就会重叠且没有任何角点落在对方的区域范围内。

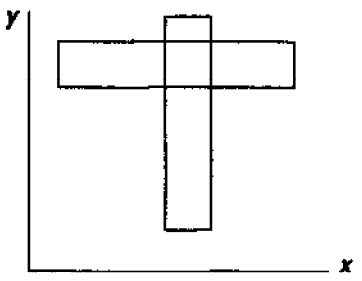



图7-3 两个彼此重叠、却没有角点落在对方区域范围内的示意图

接下来，我们来看第二种——两个矩形彼此重叠且有一个角点落在对方的区域范围内的情况，如图7-4所示。这是一种比较简单的情况，只要检查某一矩形的四个角点是否落在另一个矩形的区域范围内就能判断出它们是否发生了重叠。

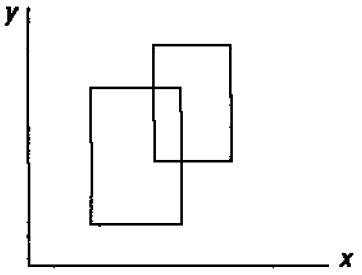



图7-4 有一个角点落在另一个矩形内的情况

第三种情况是：在重叠着的两个矩形中，其中某个矩形有两个角点落在了对方的区域范围内；也就是说，有一个矩形的半截落在了另一个矩形的区域范围内，如图7-5所示。此时，有一个矩形的四个角点都没落在对方的区域范围内，而另一个矩形却有两个角点落在了对方的区域范围内。如果你检查的是前一个矩形，就不会发现它们出现了重叠；如果你检查的是后一个矩形，那你至少要检查三个角点才能断定它们出现了这样的重叠。但你在事先是无法知道哪个矩形没有角点落在对方的区域范围内的，所以你必须检查每个矩形的三个角点才能发现它们是否出现了这样的重叠。

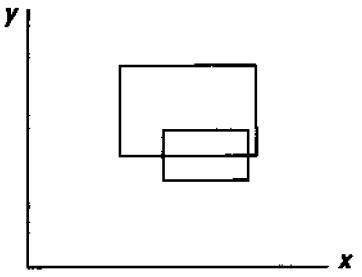



图7-5 一个矩形的两个角点落在对方区域范围内的情况

有三个角点落在对方区域范围内的情况最为简单——不可能出现这样的情况。无论你怎样画，都不可能画出一个矩形有三个角点落在对方区域范围内的情况。

有四个角点落在对方区域范围内的情况是有可能的。此时，一个矩形完全被包容在另一个矩形的里面，如图7-6所示。你只需要检查两个矩形各自的一个角点，就能判断出它们是否出现了重叠。

现在，把有0、1、2、4个角点落在对方区域范围内的这些情况归纳一下。0角点

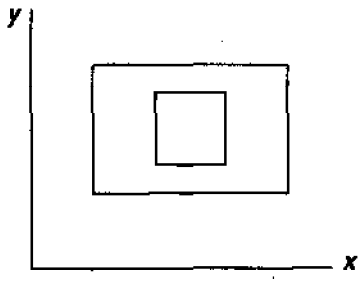



图7-6　一个矩形的四个角点落在对方区域范围内的情况

情况需要对两个矩形的宽度、高度以及位置进行检查；1角点情况需要对某一个矩形的四个角点进行检查；2角点情况需要分别检查两个矩形各自的三个角点；4角点情况需要分别检查两个矩形各自的一个角点。对这几种情况分别进行比较判断未免有些罗嗦。能不能只用一条比较语句就把这几种情况都检查过来来呢？首先，为了判断两个矩形是否出现了“0角点重叠”的情况，需要检查它们的宽度、高度、和位置。接着，为了判断两个矩形是否出现了“1角点重叠”的情况，需要检查其中一个矩形的四个角点。然后，为了判断两个矩形是否出现了“2角点重叠”的情况，还要再多检查另一个矩形的三个角点才行。最后，为了判断两个矩形是否出现了“4角点重叠”的情况，需要检查它们各自的一个角点——不过，针对“2角点重叠”情况的检查已经把“4角点重叠”情况的检查包含在内了。把这些分析总结一下，我们就得到了下面这些用来检查两个矩形是否发生了重叠的条件：

1）两个矩形的高度、宽度和位置。

2）依次检查其中一个矩形的四个角点，看它们是否有一个落在了对方的区域范围内。

3）依次检查另一个矩形的三个角点，看它们是否有一个落在了对方的区域范围内。

矩形是否重叠问题的这一解决方案是正确的，但看上去却有一种效率不高的感觉。它得检查两个矩形的高度、宽度和位置，还得检查总共八个角点中的七个——而每检查一个角点都必须进行4次比较操作。也就是说，你总共要进行34次比较操作才能得到最终的答案。

还有没有更好的解决方案呢？能不能从两个矩形彼此没有发生重叠的情况入手呢？只要把两个矩形没有发生重叠的情况弄清楚，不就等于知道了它们会在什么时候发生重叠了吗？两个矩形没有发生重叠的情况是比较容易找出来的。把这两个矩形分别叫做矩形A和矩形B。矩形A和B彼此没有发生重叠的条况可以归纳为四种：A在B的上面、在它的下面、在它的右面、在它的左面；这几种条件可能同时发生——比如A位于B的左上面。但不管怎么说，只要出现上述四种情况之一，就能判定两个矩形没有发生重叠。我们把以上这些分析总结如下。

在以下几种情况里，两个矩形将不发生重叠：

1）矩形A左上角的$x$坐标大于矩形B右下角的$x$坐标。

2）矩形A左上角的$y$坐标小于矩形B右下角的$y$坐标。

3）矩形A右下角的$x$坐标小于矩形B左上角的$x$坐标。

4）矩形A右下角的$y$坐标大于矩形B左上角的$y$坐标。

这个方案要简单得多了，只需进行4次比较和一次求反（NOT），就能判断出两个矩形是否发生了重叠。按照这一方案，我们写出了如下所示的函数代码：

```
int Overlap(struct rect A, struct rect B)
{
    return(!((A.UL.x > B.LR.x) ||
             (A.UL.y < B.LR.y) ||
             (A.LR.x < B.UL.x) ||
             (A.LR.y > B.UL.y)));
}
```

这个函数本身是没有问题的，但你可以再对它进行一下优化。能不能把那个逻辑非（NOT）操作节省掉呢？你还记得逻辑学里的德·莫甘定律（DeMorgan's Law）吗？这个定律可以被表述为：

¬(A OR B) = ¬A AND ¬B[3]（“¬”是NOT操作的逻辑运算符）[⊖]

¬(A AND B) = ¬A OR ¬B

另外，大家当然还应该知道：

$\neg(A > B)$ 等价于$(B \leqslant A)$

根据这些定律和法则，你将得到下面这样的函数：

```
int Overlap(struct rect A, struct rect B)
{
    return((A.UL.x <= B.LR.x) &&
           (A.UL.y >= B.LR.y) &&
           (A.LR.x >= B.UL.x) &&
           (A.LR.y <= B.UL.y));
}
```

为了确保自己不会有所遗漏，你应该复查一下这些条件是否有意义。这个函数将按以下规则来判断两个矩形是否发生了重叠：

1）矩形A的左边在矩形B右边的左面。

2）矩形A的上边在矩形B底边的上面。

3）矩形A的右边在矩形B左边的右面。

4）矩形A的底边在矩形B上边的下面。

当满足这些条件之一时，矩形B必定不在矩形A的区域范围之外；也就是说，它们必然会重叠。这道面试题到这里就干净利落地解答完毕了。

---

⊖ 译者注。

## 7.7 面试例题：字节的升序存储和降序存储方式

- **请编写一个函数来判断某计算机的字节存储顺序是升序（little-endian）还是降序（big-endian）。**

这道面试题的考察重点是求职者对计算机体系结构的熟悉程度和他们的编程能力。面试考官想知道你对“字节的存储顺序”这一概念是否熟悉。如果你熟悉这个概念，就应该向面试考官做出解释，至少要试着指出“升序”和“降序”的区别——哪怕你已经分不清它们。如果你不熟悉这个概念，就只能向面试考官请教了。

“字节的存储顺序”指的是多字节数据的各个字节在计算机里的存储顺序。现代的计算机几乎都采用了以多个字节来表示一种基本数据类型的做法。比如说，整数通常要用4个字节来表示。从理论上讲，用来表示整数的那4个字节可以按任何顺序来存储；但在实践中，它们最常见的存储顺序则不外乎两种：一种是按从最低位字节（least-signaificant byte，LSB）到最高位字节（most-signaificant byte，MSB）的顺序进行存储；另一种是按从MSB到LSB的顺序进行存储。这里所说的“高位字节”指的是字（word）中的高位字节。如果字节表示的是字的低值部分，我们就说它是这个字的LSB。比如说，数值“5A6C”中的LSB就是“6C”。反过来说，如果字节表示的是字的高值部分，我们就说它是这个字的MSB。比如说，数值“5A6C”中的MSB是“5A”。

在一台采用字节降序存储方案的计算机里，MSB将被保存在最低位的地址里；在一台采用字节升序存储方案的计算机里，LSB将被保存在最低位的地址里。比如说，采用字节降序存储方案的计算机将把十六进制值“A45C”中的“A4”保存在第一个字节里，把“5C”保存在第二个字节里；而一台采用字节降序存储方案的计算机将把“5C”保存在第一个字节里，把“A4”保存在第二个字节里。

解答这道面试题需要用到某种多字节数据。具体选择哪一种并不要紧，只要它是一个字节以上的就够用了。下面，我们将以整数为例来进行讨论。你必须设法检测出整数中的哪个字节是LSB，哪个字节又是MSB。如果你把整数的值设置为“1”，就可以区分出MSB和LSB了——在这个整数里，有一个字节的值是“1”，而另一个字节的值则是“0”。准确地说，LSB里的值将是“1”，而MSB里的值则是“0”。

可惜的是，对整数中的特定字节进行存取却不是一件容易的事情。大家也许已经想到了位操作符，因为它们能够让你对变量中的位进行操作。不过，它们并不像预期的那么有用，因为位操作符是按位由低到高排列的顺序来完成操作的。比如说，如果你用一个左移位操作符把一个整数向左移动8个位，左移位操作符将把这个整数当做连续的32个位而不管这个整数在计算机里的字节存储顺序是怎样的。这就使你很难利用位操作符来判断出字节的存储顺序。

怎样才能对构成整数的各个字节进行访问和操作呢？字符是一种单字节数据类型，把整数当做4个连续排列的字符似乎是个好办法。要想做到这一点，你首先得创建一个指向整数的指针，再把这个整数指针映射为一个字符指针——这样，你就能把整数当做一个元素是单字节数据类型的数组来对待了。然后，利用这个字符指针，你就能对字节进行检查并判定出计算机的字节存储顺序了。

我们把判断计算机字节存储顺序的思路总结一下：首先，创建一个整数指针，让它指向一个取值为“1”的整数。然后，把这个整数指针映射为一个“char*”指针；这样，你就能对整数中的各个字节进行访问和操作了。字符指针使你能够以字节为单位对一个4字节的整数进行操作。再往后，检查第一个字节，看它的值是否为“1”。如果这个字节的值是“1”，这台计算机采用的就是字节的升序存储顺序——因为LSB被保存在最低位的地址里；如果这个字节的值是“0”，这台计算机采用的就是字节的降序存储顺序——因为MSB被保存在最低位的地址里。我们把整个算法归纳为以下几个要点：

初始化一个取值为“1”的整数

把一个指向这个整数的指针映射为“char *”

如果字符指针处的取值是“1”，这台计算机采用的就是字节的升序存储顺序

如果字符指针处的取值是“0”，这台计算机采用的就是字节的降序存储顺序

下面是我们根据这个算法写出来的代码：

```
/* Returns 1 if the machine is little-endian, 0 if the
 * machine is big-endian
 */
int Endianness(void)
{
    int testNum;
    char *ptr;

    testNum = 1;
    ptr = (char *) &testNum;
    return (*ptr); /* Returns the byte at the lowest address */
}
```

这个解决方案已经足以满足程序设计面试的要求了。但是，解答出面试题只是求职着参加程序设计面试的任务之一，更为重要的是能否给面试考官留下一个深刻的印象。因此，你应该再努力一下，看还能不能为这道试题找出一个更精巧的解决方案来。C语言为程序员提供了各种“union”类型。这种类型与“struct”很相似，只是它所有的数据成员都是从内存中的同一个位置开始存储的。这就使你能够把同样的数据当做不同的变量类型来使用。“union”的语法与“struct”的语法几乎是一模一样的。下面是我们利用“union”类型写出来的代码：

```
/* Returns 1 if the machine is little-endian, 0 if the
 * machine is big-endian
 */
int Endianness(void)
{
    union {
        int theInteger;
        char singleByte;
    } endianTest;

    endianTest.theInteger = 1;
    return endianTest.singleByte;
}
```

## 7.8 面试例题："1"的个数

- **请编写一个函数，让它把一个给定整数的二进制表示形式中的"1"的个数统计出来。**

乍看起来，这道面试题似乎是一个数制转换问题——你得设计一个算法来把一个10进制整数转换为它的二进数形式。这其实是一个误区，因为整数在计算机里的内部存储形式已经是它的二进制补码了。所以你根本不需要进行数制转换，直接统计"1"的个数就可以了。

"1"的个数并不难统计，把各位的取值情况都测试一遍不就行了吗？那么，有没有能对指定位的取值情况进行测试的操作符呢？如果有这样一个操作符，你就能遍历所有的位并统计出它们当中共有多少个"1"来。很可惜，如此理想的操作符是不存在的。

另一条路是设计一个能够利用现有的位操作符来测试出各位取值情况的例程。我们先把注意力集中到数值的最低位上，看能不能把它的取值情况测试出来。把给定整数与"1" AND在一起会怎样？在一台8位整数的计算机（大多数现代计算机里的整数至少是32位的。但32位的整数写起来实在是太麻烦了，所以我们决定用一个8位的整数来做为有关讨论中的例子）里，"1"将被保存为"00000001"。所以，如果给定整数最低位的取值是"0"，AND操作的结果就将是"00000000"；如果给定整数最低位的取值是"1"，AND操作的结果就将是"00000001"。总得来说，只要能构造出正确的"掩码"（mask），就能检测出任何一个位的取值情况。掩码的构造规则是：把你不想检测的位全部设置为"0"，把你想检测的位设置为"1"；这样得到的整数就是你想要的掩码。当你把这个掩码与你想要检查的值做AND运算时，如果在掩码中被设置为"1"的位的运算结果为"0"，就说明被检测数值中的对应位取值为"0"；如果运算结果非零，就说明被检测数值中的对应位取值为"1"。

你可以为每一个位分别构造一个掩码，再利用这些掩码来统计"1"的个数。比

如说，第一个掩码应该是“00000001”，后续掩码将依次是“00000010”、“00000100”、“00001000”……这个思路是对的，但面试考官却未必喜欢你写出这么多掩码。这些掩码有什么变化规律呢？前后两个掩码之间的变化并不大，只是里面的“1”往左面移动了一个位置而已。也就是说，你完全用不着把这些掩码全部一次性地构造出来——你可以用左移位操作符来依次地构造出它们。你只需从掩码“00000001”开始，再把这个掩码依次向左移动一个位，就能生成出所有的掩码了。这是一个好办法，如果能分析到这一步，你的解决方案应该是可以接受的了。但是，还存在着另外一个更快、更好的解决方案，它只需要使用一个掩码。

只有一个掩码，你能干些什么呢？你必须对给定整数中的每一个位进行检查，所以每次循环都必须掩住一个不同的位。其实你已经做到了这一点——对掩码进行左移位，给定的整数则保持不变。根据同一原理，如果你保持掩码不变而对那个给定的整数进行左移位，不就能用同一个掩码来检测所有的位了码？最容易想到的掩码仍是“00000001”，它本身是用来检测最低位的。如果反复对给定整数进行右移位的话，迟早会把每一个位都移动到最低位来。我们不妨以“00000101”为例来实验一下。它的最低位是“1”，所以我们要给统计值加上1，然后再对这个整数进行右移位，得到“00000010”。这一次，最低位是“0”。继续做右移位，得到“00000001”。最低位是“1”，统计值递增为2。再进行右移位将得到“00000000”。当整数的值变为“0”——也就是没有可移位的东西时，你就可以停止进行统计了。就这个例子而言，用不着把所有的位都遍历过来就能统计出所有的“1”来；所以与前面那个使用了好多个掩码的算法相比，这个算法的效率应该是很高的。根据以上分析，我们把这个算法总结为以下几个要点：

```
从count=0开始
当整数number不为零时，循环
    如果number AND 1 = 1，递增count
    把整数int右移一个位
返回统计值count
```

最后，考虑这个算法的特例情况，你需要检查这个算法对正整数、负整数和零的处理情况。如果给定整数是“0”，这个算法将立刻正确地返回一个统计值0，表明二进制表示形式里没有“1”。现在，来看看它在遇到负数时的执行情况。在做右移位操作的时候，空缺将被填补为符号位，可负数的符号位是“1”而不是“0”。也就是说，给定负整数的全体位迟早会都变成“1”而不是“0”。要想解决这一问题，就需要把给定整数的二进制表示形式当做一个无符号数来处理。这样，移位操作符将在不再用符号位来填补空缺。既然新增加的位都将用“0”而不是“1”来填补，给定整数的位迟早会全都变成“0”。最后，考虑一下这个算法对正整数的处理情况。

我们在推导算法时使用的就是一个正整数例子，执行过程中没有出现问题。

这个算法的实现代码如下所示：

```
int NumOnesInBinary(unsigned int number)
{
     int numOnes = 0;
     while (number) {
          if (number & 1)
              numOnes++;
          number = number >> 1;
     }
     return numOnes;
}
```

这个函数的执行时间是怎样的？这个函数要通过while循环把所有的“1”都统计出来。在最佳情况里，给定整数是0，这个函数不会进入while循环。在最坏情况里，这个算法的执行时间将是O($n$)，而$n$则是整数的位长度。

除非你是二进制位操作方面的专家，否则，这个算法就将是你在程序设计面试中能够拿出来的最好的东西。可我们要告诉你的是，还存在着一个更好的算法。请继续往下看。如果你用给定整数减去1，这个整数的位会发生什么样的变化呢？这些变化又有什么规律呢？在减去1之后，给定整数中最右边的“1”以及它后面的位都将被翻转。比如说，用整数“01110000”减去1将得到“01101111”。

如果把给定整数减去1后再与它原来的值进行AND操作，我们就将得到一个新的整数；新整数的位与原来的整数基本一致，只有最右面的“1”变成了“0”。比如说，01110000 AND (01110000 – 1) = 01110000 AND 01101111 = 01100000。

在给定整数变成0之前，上述过程能重复进行多少次？数一下就知道了，它恰好就是该数字的二进制表示形式中的“1”的个数。我们把这个算法总结为以下几个要点：

从count=0开始

当整数number不为零时，循环

number AND (number – 1)

递增count

返回count

这个算法的实现代码如下所示：

```
int NumOnesInBinary(int number)
{
     int numOnes = 0;
     while (number) {
          number = number & (number - 1);
          numOnes++;
     }
```

```
    return numOnes;
}
```

这个算法的执行时间是O($m$)，$m$是给定整数的二进制表示形式中的“1”的个数。也许还有更好的算法也说不定。注意，这个算法只是我们为帮助大家开拓思路而提供的。我们在前面给出的解决方案应该能够应付程序设计面试了。

## 7.9 面试例题：简单的SQL查询

- 给定一个如下所示的数据库表：

```
Olympics(city CHAR(16), year INT(4));
```

请用一条SQL语句把1976年举办奥运会的Montreal添加到数据库里去。

这道极其简单的试题能够让面试考官一眼看出谁真的有SQL使用经验，谁又在编造求职简历。如果你熟悉SQL，就会知道它的答案将是一条毫无技巧可言的“INSERT”语句。如果你不熟悉SQL，那还是老老实实地承认比较好。这个问题的正确答案是：

```
INSERT INTO Olympics VALUES('Montreal', 1976);
```

## 7.10 面试例题：公司和员工数据库

- 给定一个数据库，里面有两个如下所示的数据表：

```
Company(companyName CHAR(30), id INT(4));
EmployeesHired(id INT(4), numHired INT(4),
    fiscalQuarter INT(4));
```

假定输入参数fiscalQuarter（意思是“财季”）的可取值都是合法的，即从1到4。这个数据库的样本数据如表7-3所示。

表7-3 Player表的样本数据

| COMPANYNAME（公司名称） | ID（编号） |
|---|---|
| Larry Smith | 6 |
| David Grozalez | 9 |
| Geroge Rogers | 19 |
| Rajiv Williams | 3 |

| ID（编号） | NUMHIRED（招聘人数） | FISCALQUARTER（财季） |
|---|---|---|
| 3 | 3 | 3 |
| 9 | 2 | 4 |
| 19 | 4 | 1 |
| 6 | 2 | 1 |

**现在，请用一条SQL语句把在第4财季招聘过员工的公司名称全都查出来。**

这道面试题需要用两个表来检索数据。你需要把两个表关联起来才能找到你需要的数据。id字段是这两个表惟一的共用键（common key），所以你必须用id来关联这两个表。把两个表关联起来之后，你就可以把fiscalQuarter等于4且招聘过员工的公司名称查出来了。最终的SQL语句应该是下面这样的：

```
SELECT companyName FROM Company, EmployeesHired
WHERE Company.id = EmployeesHired.id AND fiscalQuarter = 4;
```

这条SQL语句有一点儿小毛病：如果有家公司没在第4财季招聘过员工怎么办？比如说，如果数据库里有一条EmployeeHired（6，0，4）这样的记录项，会发生什么事情？虽然没有在第4财季招聘过任何员工，但编号为6的公司仍会出现在上面这条SQL语句的查询结果中。这个小毛病很好纠正，只要保证numHired必须大于0就行了。下面是改正后的SQL语句：

```
SELECT companyName FROM Company, EmployeesHired
WHERE Company.id = EmployeesHired.id AND fiscalQuarter = 4 AND numHired
> 0;
```

- **现在，仍使用刚才的表，请用一条SQL语句把从第1到第4财季没有招聘过任何员工的公司名称全都查出来。**

解答这道试题的最佳路线是从上一道例题的答案出发。你已经知道如何查出在第4财季招聘过员工的公司名称了。如果去掉WHERE条件中的“fiscalQuarter = 4”，就能知道有哪些公司在第1到第4财季期间招聘过员工；而没有出现在这个查询结果里的公司就将是这道面试题的答案。如果还想再稍微做点优化的话，你可以对表EmployeeHired中的id进行查询，那些id没有出现在EmployeeHired数据表里的公司就是我们想要的答案。最终的查询语句如下所示：

```
SELECT companyName FROM Company WHERE id NOT IN
(SELECT id from EmployeesHired WHERE numHired > 0);
```

- **最后，请把在第1到第4财季期间招聘过员工的公司名称和它们各自招聘的员工总人数查出来。**

既然需要统计总人数，你就得使用SQL中的“SUM”统计功能了。这道面试题并不是让你去统计整个numHired数据列的总和，它只要求你把id相同的数值统计在一起；所以你还要利用“GROUP BY”功能才能把同属一组的数据总和统计出来。除增加了“GROUP BY”子句以外，这个查询与去掉了“WHERE”子句中的“fiscalQuarter = 4”条件后的第一道SQL面试题的答案很相似。下面就是这个查询：

```
SELECT companyName, SUM(numHired)
FROM Company, EmployeesHired
WHERE Company.id = EmployeesHired.id
```

```
GROUP BY companyName;
```

## 7.11 面试例题：最大值，不允许使用统计功能

**• 给定一个如下所示的SQL数据库工作区：**

```
Test(num INT(4));
```

**请用一条SQL语句返回num的最大值，但不许使用统计功能（如MAX、MIN等）。**

这道面试题等于是让你绑着脚跳舞——让你去找一个最大值，却又不让你用专为这一目的而提供的功能。在解答这类问题时，先画出有关表并填上一些样本数据不失为一个好的出发点。表7-4就是我们画出来的样本表。

就这个样本表而言，你需要把23找出来，因为表中的其他元素都小于23。虽然原理正确，可这种说法对构造符合题目要求的SQL语句并没有多大的帮助。换个角度看，23是这个表中惟一没有比它还大的那个元素，这种说法有那么点意思了。如果你的SQL语句能够把没有比它还大的那些个元素找出来，那你的查询结果肯定只有23这个数，而你也就解决了这一问题。下面，我们就来看看怎样才能构造出一条能够找出满足这一条件的SQL语句。

首先，你需要把还有比它们更大的那些元素都找出来。这个查询我们还是能设法写出来的：把num表与它自己关联起来并创建出一个这样的数据表：新表有两个数据列，其中一个数据列的值小于另一个数据列中的对应值,如下所示：

```
SELECT Lesser.num, Greater.num
FROM Test AS Greater, Test AS Lesser
WHERE Lesser.num < Greater.num;
```

用这个查询对表7-4中的样本数据进行处理，我们将得到如表7-5所示的查询结果。

**表7-4 num表及样本数据**

| NUM |
|---|
| 5 |
| 23 |
| -6 |
| 7 |

**表7-5 临时表**

| LESSER | GREATER |
|---|---|
| -6 | 23 |
| 5 | 23 |
| 7 | 23 |
| -6 | 7 |
| 5 | 7 |
| -6 | 5 |

很明显，除最大值23以外，其他数值都在“LESSER”数据列至少出现过一次。这就给我们一个提示：如果能把没有在“LESSER”数据列里出现过的数值选出来，那它不正是我们想要的东西吗？这样，我们就得到了如下所示的查询：

```
SELECT num from Test WHERE num NOT IN
(SELECT Lesser.num FROM Test AS Greater, Test AS Lesser
WHERE Lesser.num < Greater.num);
```

这个查询还有一点小毛病。如果最大值在Test数据表里重复出现，那它就将被返回两次。为了避免这种情况，你应该再增加一个保留字“DISTINCT”。下面是这道面试题的最终解答：

```
SELECT DISTINCT num from Test WHERE num NOT IN
(SELECT Lesser.num FROM Test AS Greater, Test AS Lesser
WHERE Lesser.num < Greater.num);
```

## 7.12 面试例题：生产者/消费者问题

- **请编写一个Producer线程和一个Consumer线程，两个线程共享着一个固定长度的缓冲区和该缓冲区上的一个读写索引index。Producer负责把一些随机数放到缓冲区里，Consumer则负责删除那些随机数。请用C语言中的信号量和Java中的线程方法来实现这一问题。**

如果你曾经编写过多线程程序，那你可能早就见过这个问题。这是一个非常有代表性的并发问题。如果你没有多线程程序的设计经验，要想把这道面试题解答好可就没那么容易了。

我们先用非并发技术来尝试着实现一下这个问题，看它存在着什么样的弊病。在没有并发控制的情况下，这个算法并不难实现。Producer和Consumer函数的代码如下所示：

```
static int index = 0;
static int buffer[8];
void Producer()
{
    while (1) {
        if (index < 7) {
            buffer[index] = rand();
            index++;
        }
    }
}

void Consumer()
{
```

```
    while (1) {
        if (index > 0) {
            printf("%d\n", buffer[index]);
            index--;
        }
    }
}
```

这个实现中的主要弊病是没有对缓冲区进行保护。因此而可能出的问题之一是：如果Producer在index尚未更新之前把随机数写入了缓冲区，就会覆盖掉原来的内容，让Consumer读到错误的数据。此外，对index位于缓冲区最后一个元素位置上时的更新操作考虑不周，扰乱Producer和Consumer的配合。

这段代码的关键操作包含了数组的访问和index的更新。你必须用一个二元信号量来保护它们——每次只允许一个线程去执行这几条语句。因为这是一个二元信号量，所以你必须把它初始化为1。修改后的代码如下所示：

```
static int index = 0;
static int buffer[8];
static Semaphore bufferWrite;

void init()
{
    bufferWrite = newSemaphore(1);
}

void Producer()
{
    while (1) {
        wait(BufferWrite);
        if (index < 7) {
            buffer[index] = rand();
            index++;
        }
        signal(BufferWrite);
    }
}

void Consumer()
{
    while (1) {
        wait(BufferWrite);
        if (index > 0) {
            printf("%d\n", buffer[index]);
            index--;
        }
        signal(BufferWrite);
```

```
    }
}
```

这一解决方案纠正了前面的弊病。共享变量全都得到了保护，线程的工作情况也正确了。但纠正错误只是一个方面，你还应该考虑到执行效率的问题。比如说，如果Producer比Consumer慢很多，会发生什么情况？Consumer将频繁地苏醒，获得信号量，可缓冲区里没有东西，它于是又再次进入休眠等待状态。这种反复“苏醒、什么也不做、再休眠”的情况叫做“忙等待”（busy waiting），是对资源的一种浪费。原则上讲，只有在有东西可消费的时候才应该让Consumer苏醒过来。类似地，只有在缓冲区里有空缺的时候才应该让Producer苏醒过来。

效率低下也是一种bug，你可以用信号量技术来纠正和改善之。我们需要一个信号量来表明缓冲区是否已经满了。Producer线程要靠这个信号量来唤醒；而Consumer线程每从缓冲区里取走一个元素，就触发一次这个信号量。起初，缓冲区是空的，所以这个信号量应该被初始化为缓冲区的长度，允许Producer往里面放入元素；等缓冲区不再有空缺时（此时，这个信号量将等于0），这个信号量就会阻塞Producer线程。

类似地，我们还需要一个信号量来表明缓冲区是否为空。Consumer线程要靠这个信号量来唤醒；而Producer线程每往缓冲区里放入一个元素，就触发一次这个信号量。起初，缓冲区是空的，没有可消费的元素，所以这个信号量应该被初始化为0以阻塞Consumer线程。等缓冲区里有元素后（此时，这个信号量将大于0），这个信号量就会唤醒Consumer线程。

最后，一定要把这些信号量在程序代码中的先后顺序安排好。如果用来保护关键代码的二元信号量（即代码中的bufferWrite信号量）出现在用来改善执行效率的信号量之前（即代码中的bufferNotFull和bufferNotEmpty信号量），那么，当Consumer线程拥有着bufferWrite信号量却必须等待bufferNotEmpty信号量时，就会出现死锁局面。类似地，当Producer线程拥有着bufferWrite信号量却必须等待bufferNotFull信号量时，也会出现死锁局面。因此，用来保护关键代码的二元信号量必须出现在其他信号量之后。

把这些事情都安排妥当之后，我们就得到了如下所示的代码：

```
static int index = 0;
static int buffer[8];
static Semaphore bufferWrite;
static Semaphore bufferNotFull;
static Semaphore bufferNotEmpty;

void init(void)
{
    bufferWrite = newSemaphore(1);
```

```
    bufferNotFull = newSemaphore(8);
    bufferNotEmpty = newSemaphore(0);
}

void Producer(void)
{
    while (1) {
        wait(bufferNotFull);
        wait(bufferWrite);
        buffer[index] = rand();
        index++;
        signal(bufferWrite);
        signal(bufferNotEmpty);
    }
}

void Consumer()
{
    while (1) {
        wait(bufferNotEmpty);
        wait(bufferWrite);
        printf("%d\n", buffer[index]);
        index--;
        signal(bufferWrite);
        signal(bufferNotFull);
    }
}
```

Java对并发线程的处理办法与C语言稍有不同。一切Java线程都必须对Thread类进行扩展。它们应该实现有自己的run方法，这个方法将在线程启动时被调用。在下面的代码里，我们还使用了Java的保留字“synchronized”、“wait”方法、和“notifyAll”方法。为了让大家对线程函数的调用方法有所了解，我们把主函数main()也收录在代码中了。

```
import java.util.Random;

class Producer extends Thread {
    private static final int MAX_CAPACITY = 8;
    private static final int RANDOM_RANGE = 128;
    private int[] buffer;
    private int index;
    private Random generator;

    public Producer()
    {
        buffer = new int [MAX_CAPACITY];
        generator = new Random(23);
```

```
        index = 0;  // initally empty
    }

    public void run()
    {
        while (true) {
            try {
                putInt();
            }
            catch(InterruptedException e) {}
        }
    }

    private synchronized void putInt() throws InterruptedException
    {
        while (index == MAX_CAPACITY) { // Buffer is full.
            wait();
        }
        buffer[index] = generator.nextInt(RANDOM_RANGE);
        index++;
        notifyAll();  // Let other threads know that something
                      // has happened.
    }

    // Called by the consumer.
    public synchronized int getInt() throws InterruptedException
    {
        notifyAll(); // Need to make sure that we're
                     // not stuck with this thread.
        while (index <= 0) {
            wait();
        }

        index--;
        return buffer[index];
    }
}


class Consumer extends Thread {
    private Producer producer;

    public Consumer(Producer theProducer)
    {
        producer = theProducer;
    }

    public void run()
```

```
    {
        try {
            while (true) {
                System.out.println("Int is " + producer.getInt());
            }
        }
        catch (InterruptedException e) {}
    }
    public static void main(String args[]) {
        Producer producer = new Producer();
        producer.start();
        new Consumer(producer).start();
    }
}
```

# 与计数、测量、排序有关的智力题

在程序设计面试中，除技术和程序设计方面的问题外，求职者还经常会遇到一些智力题。这些智力题多偏重于数学和逻辑，与计算机没有什么直接关系。有些面试考官认为这些问题很愚蠢，不能反映出求职者的实际工作能力，也就不会出这类问题。但也有不少面试考官喜欢用智力题来考察求职者解决问题的能力——也许是最重要的工作本领。还有一些面试考官，因为听说微软公司等行业巨头在面试中使用了智力题，于是也向求职者提出这类的问题。不管出于什么样的原因，在某些面试中，几乎有三分之一的问题是智力题。

我们认为，智力测验确实能体现出求职者解答数学难题的能力，但与他能否成为一名有价值的员工却并没有多大的关系。本章和下一章里的讨论主要是想让大家对这类难题有一个了解，使大家在智力测验中能够有备而来。与程序设计和技术性问题相比，这些问题的取材范围更加广泛，所以提前复习并没有多大的必要。不过，智力难题也有它自身的特点，熟悉这些特点和常见的题型肯定会对你有很大的帮助。

智力难题最重要的一个特点是它的答案很少有直白或者明显的。在面试过程中，程序设计或技术部分的问题主要是为了考察求职者对有关概念的熟悉程度，所以题目本身往往比较简捷明了。智力题则不同，它们的目的是为了考察求职者解决问题的能力，所以不经过一番思考和努力是得不出答案的。因此，那些立刻就能想到的答案要么是不正确的，要么不是最好的。比如说，有这样一个问题："你从山脚坐缆车上山，当你到达山顶时，从你身边经过的缆车车厢有多少？"大多数人都会立刻想到是所有车厢的一半。这个答案很明显，也有一定的道理：上山和下山的车厢各有一半，只有下山的车厢才会从你的身边经过。可这个答案是错的——因为两旁边的车厢都在移动，所以从你身边经过的车厢应是所有其他车厢。（如果你没有从头坐上缆车或者没有坐到头，从你身边经过的车厢就会少那么几辆。）

当你遇到只可能有两种答案（比如“是或否”之类）的智力题时，下面这一规律对你将特别有帮助：你最先想到的答案极有可能是错误的。“答案肯定是‘是’。因为如果答案是‘否’的话，这个问题就太简单了，你也就用不着拿它来考我了”——这句话你不能对面试考官说，但用它来指导自己的思想却会很有收获。

**提示：** 在解答智力题的时候，最容易想到的答案往往是不正确的。

智力题的另一个特点是：虽然不容易解答，但正确答案本身往往都很简单，很少会用到三角学以外的知识，也用不着做大量或者复杂的计算。如果你写在纸上的计算有好几页，就应该提醒自己是否已经走错了路；用微积分或者很难计算的数字则往往预示着你与正确答案的距离正越来越远。

这些问题之所以很难解答，是因为它们几乎都利用了人们的一些心理误区，而这些误区只会让你得出一个错误的答案。因此，你应该在不做任何假设的情况下去寻求答案。可惜的是，这一点很难做到——有时候，你在阅读题目的时候就会不自觉地做出一些假设来。比如说，当你看到“怎样才能在一个正方形的箱子的底部尽可能多地摆放一些橘子？”这道题的时候，你会立刻做出这样一些假设：橘子是小圆形的水果，它们的大小都差不多，“在箱子的底部”意味着与箱底直接接触，橘子要保持完整（不能把它们剥开后倒在箱子里）。这些假设说出来会让人觉得很奇怪——它们都是明显的事实，而且都是正确的假设。问题的关键是：（1）假设来源于人们的常识或者思维；（2）在解答问题的时候，你很难完全摆脱假设的影响。

接着刚才“摆橘子”的例子往下说。你仍继续做着假设：这是一个二维空间问题，可以用一个圆和一个正方形来分别代表橘子和箱子，必须有规律地摆放橘子才能放得最多。根据这些假设和“在平面上，把圆摆成蜂窝状将占用最小的面积”的知识，你得出了自己认为最佳的结论：把橘子规则地摆放成蜂窝状。可根据橘子和箱子的大小，你的结论却可能是错误的。

不受假设的影响是不太可能的，但在得出结论之前，你应该先把这些假设写出来并逐条地加以分析才对。在写出这些假设的时候，你应该把它们划分为三大类：几乎完全正确、可能正确、可能不正确。在进行分析的时候，你应该先着手于那些可能不正确的假设，看能不能去掉这些假设；每去掉一个假设，就重新对智力题进行一次研究。需要提醒大家的是，因为智力题都是一些很古怪的问题，所以那些定义性的假设基本上都是正确的。

我们继续分析“摆橘子”的例子。把“橘子是小圆形的水果”、“它们必须保持完整”、“与箱底接触”等几个假设划分到“几乎完全正确”那一类应该是合乎情理的。“这是一个二维空间问题，可以用一个圆和一个正方形来分别代表橘子和箱子”的假设应该划分到哪一类里去呢？圆形的橘子摆放在正方形的箱底，所以这应该是一个在平面上摆放圆的问题——虽然缺乏充足的证据，但把这条假设归入“可能正确”

那一类应该不会有什么问题。“橘子必须有规律地摆放”这条假设就比较难办了，它看起来很合理，在一个无限平面上它确实也成立。箱底与平面有一定的相似性，但能否保证这条假设成立却不好说。原则上讲，如果你觉得某条假设好像成立却又解释不了为什么，就应该提高警惕——它很可能是一条不正确的假设。因此，把“橘子必须有规律地摆放：这条假设归入“可能不正确”那一类比较保险一些。事实上，这条假设确实是不正确的。在很多场合，“摆橘子”问题的最佳解决方案是“把大多数橘子按一定规律摆放，再把剩下来的一些小橘子见缝插针地摆放”。总之，当你找到了一个你认为是惟一合乎逻辑的解决方案却被告知它是不正确的时候，回过头来对你所做的假设逐条地进行分析将是一个非常有用的策略。事情往往会是这样的：你的推理过程本身完全符合逻辑，只可惜它是建立在一条有漏洞的假设的基础上。

**提示：** 如果看起来符合逻辑的解决方案其实却是错误的，那你很可能做出了一条错误的假设。在遇到这种情况时，你应该对有关假设进行分类并逐条地加以分析，把错误的假设找出来。

人们最怕遇到的问题有两种，其中之一是极其复杂或困难、以至于你很难找出它们的解决办法——甚至连从什么地方开始下手都很难决定——的问题。不要被这种问题吓倒。你不必非得找到通往正确的解决方案的道路后才开始动手——事情也许会随着你的进展而逐渐明朗起来。如果你能从中分离出一个子问题，那么，即使你不能确定它对整个问题的解决有没有用，也应该先设法把这个子问题解决掉。试着对问题做一些简化，然后从这个简化了的问题入手去寻找答案——你可能会在解决简化版问题的时候发现一些有价值的线索。如果题目涉及到了一些步骤，请按这些步骤来试验一些比较简明的例子；这有助于你发现其中的规律，进而帮助你解决整个问题。总之，你应该不停地说话、不停地思考、不停地动作。开动脑筋积极思考肯定要比你傻待在起跑线上祈求奇迹出现更有助于解决问题。即使你没有多大的进展，你积极主动的进取行为也会给面试考官留下更深刻的印象——有谁会愿意聘用一位遇到难题就望着天花板发呆的员工呢？记住，你参加面试的目的是为了证明你能成为一名有价值的员工。能够完满地解答出面试题当然更好，可即便你没有解答出来，耐心细致地分析问题和尝试各种方法也同样能展示出你的能力和价值，也同样会得到面试考官的正面评价。

**提示：** 不要被问题的复杂性吓倒。你应该从它的子问题、简化版或各种例子入手。要有耐心，不停地思考，不停地说话。

另一类让人望而生畏的题目则恰好相反：它们非常简单或者限制条件非常严格，在给定的条件下，似乎根本没有办法去解决它们。在遇到这类问题的时候，集思广益是最佳的解决之道。试着把已知条件所允许的所有办法——包括那些看起来没多大

用处、甚至是背道而驰的办法——全都列举出来。如果题目涉及到了一些物体，请思考每一个物体、每一个物体的特性、你可以用它们来做些什么、它们又会有什么样的反应或变化，等等。如果在这类题目上卡了壳，请反思自己是否疏忽了题目所允许的某些东西。你应该把题目已知条件所允许的所有动作列成表格——你疏忽的那一点应该就隐藏在其中。列举所有的可能性往往要比你钻在牛角尖里想自己疏忽了哪些东西来得更容易。在列举各种可能性的时候，请不要光在脑子里想，一定要把它们说出来或者写在纸上。这一方面能够让面试考官知道你正在做些什么，另一方面则有助于你做得更系统、更全面彻底。

**提示：** 如果你在一个简单的题目上卡了壳，请把所有的可能性办法全都列举出来，看自己是否有所遗漏或疏忽。

还有一类问题值得在这里讨论一下。这就是所谓的估算类考题，即让你通过一个合理的过程估算出一个你并不知道的数据。这类问题很少出现在纯技术类职位的面试中，但在管理或营销类职位的面试中却比较常见。有一道很出名的估算题："请估算全美国有多少家加油站？"这道题据说是出自微软公司，撇开这些传说不论，这的确是一道有代表性的估算题。

估算题并不一定比其他类型的智力题更难。面试考官并没有指望你给出的答案就是千真万确的数据或者事实，他们想看的是你如何根据你所知道的事实推算出一个合理结果的过程。既然是进行估算，就应该把那些大额数字估算为10的乘方（至少是10的倍数），这将大大简化有关计算的复杂程度。

就拿"加油站"问题做例子好了。你的估算过程可能是这样的："给我自己的汽车加满油大约需要6分钟的时间。我平均每星期去加一次油，并且差不多每次都能在加油站看到另外两辆汽车。如果把这些情况当做全美国平均情况的话，那么每个加油站每小时就要给30辆汽车加油。假设加油站每天开放12小时，每星期开放7天，那么每星期总共就要开放84个小时。不过，很多加油站每天开放不止12个小时，所以我不妨把加油站每星期的开放时间估算为100小时。也就是说，每个加油站每星期要给3000辆汽车加油。全美国的人口总数大约是2亿5千万。因为不可能每个人都有一辆汽车，所以我把美国的汽车总数估算为1亿辆。如果别人也像我一样平均每星期去给汽车加一次油、而每个加油站每星期只能服务3000辆汽车的话，全美国就应该有大约33 000座加油站。"这个数字可能会与实际情况有较大的偏差，但从整体规模上讲却不会有太大的出入（全美国的加油站很可能在3 300到33 000座之间）。你证明了自己有能力去建立一个估算框架并迅速估算出一个合理的数字，这比你估算出来的数字是否百分之百准确更重要。如果你还有点意犹未尽的话，请试着去估算出全美国幼儿园老师的总人数、地球的周长或者一艘轮船的重量。

## 8.1 面试例题：开锁

- **在一条长长的走廊上依次排列着100把锁着的锁头。你从把这100把锁全都打开开始（第1遍）。然后，你把所有序号是2的倍数的锁头再锁上（第2遍）。接下来，你依次走到所有序号是3的倍数的锁头前，如果它是打开的，就把它锁上；如果它是锁上的，就把它打开——我们把这称为“切换锁头的状态”（第3遍）。你继续像这样在第*n*遍去切换所有序号是*n*的倍数的锁头的状态。当进行到第100遍时，你将只切换第100把锁头的状态。请问，在如此这般地进行了100遍切换之后，有多少锁头是打开的？**

**如果走廊里依次排列着*k*把锁头，那么在第*k*遍之后，有多少锁头是打开的？**

这个问题看起来很吓人。你肯定没有时间去画出100把锁并根据题目要求人工地把它们切换100遍。即使有这个时间，照这样来解决问题也未免笨拙了点——根本没有技巧可言嘛。这道题肯定会有一个更巧妙的解决办法。你的任务就是把这个更巧妙的办法找出来。

如果只是坐在那儿盯着这道题发呆，那你肯定不会想出更巧妙的解决方案来。该怎么办呢？用“暴力”来解决整个问题当然不现实，但把锁头的数目大幅减少后却还是值得一试的——说不定能找出点规律，然后再根据那个规律把整个问题解开呢。

先任选几把锁头，比如说第12把，看操作它有什么规律。你会在哪些遍里切换这第12把锁头呢？首先，在第1遍里，因为你要把所有的锁头都打开；其次，在第12遍里，因为你将从第12把锁头开始进行切换。第12遍以后的事情就用不着操心了，因为切换工作将从第12把锁头之后的位置开始。因此，你只需对这把锁头在第2遍到第11遍之间的变化情况进行排列分析就行了。下面是你的排列结果：2，4，6，8，10，12（第2遍，打开）；3，6，9，12（第3遍，锁上）；4，8，12（第4遍，打开）；5，10，15（第5遍，没碰上）；6，12（第6遍，锁上）；7，14（第7遍，没碰上）……看出什么规律了吗？没错，第12把锁头只有在遍数是数字12的因子时才会被切换。这是一个合理并且正确的推测。*n*个一组*n*个一组地数，如果能碰到12，那肯定是这个*n*自己加自己整数次才到达的12。这其实是“*n*是12的因子”的另一种说法。这个结论虽然简单，但在经过上述分析之前，它肯定不那么容易让你一眼就看出来。

12的因子有1、2、3、4、6、12；相应地，第12把锁头的状态将依次为“打开”、“锁上”、“打开”、“锁上”、“打开”、“锁上”；所以第12把锁头的最终结局将是“锁上”。这样看来，这道题似乎与自然数的因子有关。素数是只有两个因子（一个是1，一个是它本身）的自然数。对素数进行分析也许能帮我们找到一个具有普遍意义的算法来。我们选择17做为素数的代表。自然数17的因子只有1和17，所以第17把锁头

的状态依次是“打开”、“锁上”，所以它的最终结局——与第12把锁头一样——是“锁上”。很明显，就这道题目来说，素数与非素数的结局好像没有什么区别。

锁头的状态到底有什么变化规律没有？全体锁头最初都处于“打开”状态，以后的变化规律是“打开”、“锁上”依次交替。因此，当一把锁头在第2次、第4次、第6次……被切换时，它将被锁上——换句话说，如果一把锁最终被切换了偶数次，那它的结局就将是“锁上”；如果这把锁最终被切换了奇数次，那它的结局就将是“打开”。现在，我们已经知道：任何一把锁头在第1遍里都会被打开；每当遍历次数是它序号的一个因子时，它的状态也就会被切换一次。因此，如果一把锁头的最终结局是“打开”，那它肯定只有奇数个因子。

现在，我们把这个任务缩小为“在1到100之间的数字里，哪些数字有奇数个因子？”我们刚才分析过的那两个数字都有偶数个因子（多试几个例子就会发现，1到100之间的大部分数字都有偶数个因子）。为什么会这样呢？“数字‘$i$’是数字‘$n$’的一个因子”这句话到底意味着什么呢？意味着数字“$i$”乘上另外一个数字将等于“$n$”——这可是大家都知道的事情。根据乘法交换律，如果$i \times j = j \times i$，那么数字“$j$”也将是“$n$”的一个因子。也就是说，数字的因子通常是成对出现的，所以某个数字的因子个数也往往是偶数。如果某个数字的因子不成对出现，那它就会有奇数个因子，你也就能把最终处于“打开”状态的锁头找出来了。我们知道，乘法是一种二元运算，必须用到两个数字，因此通常会对称地产生两个彼此不相等的因子。那么，如果这两个数字相等（即$i = j$）会怎么样呢？依然是对称地产生了两个因子，但这两个因子却是相等的——在效果上相当于只是一个因子，所以因子的总数就会是一个奇数。此时，$i \times i = n$；即数字“$n$”将是一个平方数。赶快找个平方数，比如说16，来试试：它的因子是1、2、4、8、16；第16把锁头的状态将依次为“打开”、“锁上”“打开”、“锁上”“打开”——与我们预想的情况一样，这把锁头的最终结局是“打开”。

根据上述分析，我们得出了这样的结论：序号为平方数的锁头的最终结局将是“打开”。1到100（包括1和100在内）之间的平方数有1、4、9、16、25、36、49、64、81、100等总共10个；所以在经过100遍切换之后，将有10把锁头是处于“打开”状态的。

类似地，对$k$把锁头的普遍情况来说，最终结局是“打开”的锁头的个数就等于1到$k$之间（1和$k$也包括在内）的平方数的个数。那么，怎样才能最快地把平方数的个数统计出来呢？平方数的分布情况不均匀，不便于统计；但大于0的平方数的平方根却能构成了一个自然数序列，很容易统计：序列中的最后一个数就是这个序列中的数字的个数。比如说，（1、4、9、16、25）的平方根是（1、2、3、4、5）；最后一个平方根是5，而平方数和平方根的个数也正好是5个。于是，你只要把等于或小于数字“$k$”的最大平方数的平方根找出来，就能知道$k$把锁头在经过k遍切换后仍处

于“打开”状态的锁头到底有多少个。

如果$k$本身就是一个平方数，事情就简单了；但它并不见得是一个平方数。此时，$k$的平方根将是一个实数而非整数；如果你把这个实数向下舍入为与它最接近的整数，这个整数的平方就将是小于k的最大平方数——恰好是我们想要的东西。把实数向下舍入为与它最接近的整数的操作叫做“求底”（floor）。因此，就总共有$k$把锁头的普遍情况而言，在经过$k$遍切换之后，将有“floor(sqrt($k$))”把锁头仍处于“打开”状态。

解答这道面试例题的关键是：用一个子问题——虽然我们对解决这个子问题是否有助于解决整个问题并没有多大的把握——来简化分析过程。有些尝试——比如对素数的分析——没起到多大的帮助作用，但其他尝试却帮助我们找出了解答这道面试例题的规律。在面试过程中，哪怕出现了最不理想的局面——各种尝试都无助于你找到最终的解决方案，像上面这样的表现也会给面试考官留下一个良好而又深刻的印象：不畏困难，不断进取，努力寻找正确方向。

## 8.2 面试例题：三个开关

- **走廊的另一头有一个房间，房间里有三盏关着的白炽灯。你站在走廊的这一头，身边的墙上有三个开关，每个开关控制着走廊另一头的一盏白炽灯。从你站的位置看不到灯光。现在，请设法把开关与灯的对应关系找出来——只允许你进入有灯的那个房间一次。**

这道例题的难点很明显：每个开关有两个状态（开或关），可需要你判断的灯却有三盏。如果把三个开关中的某一个扳成与另外两个开关不一样的状态，就能轻易地判断出它对应着哪一盏灯。但你怎样判断另外两个开关和另外两盏灯的对应关系呢？

要想把例题中的不可能变成可能，就必须在给定的已知条件上做文章。这道题的关键似乎是开关和灯泡。关于开关和灯泡你又知道些什么呢？开关是接通和断开电路用的。接通开关，电流就会经过它。灯泡是一个装有电热丝的玻璃泡，当电流流经电热丝时，灯泡就会把电能转换成光和热。

如何利用这些事实来解答这道例题呢？哪些东西是你能够测量和感受的呢？开关的特性好像帮不上什么忙，当然了，看开关还是要比测量电流来的简单。从灯泡入手好像希望大点。你可以用眼看到它们发出来的光，用手感受到它们发出来的热。灯泡会不会发光要完全取决于开关的状态——开关接通，灯泡发光；开关断开，灯泡熄灭。灯泡发出来的热有什么利用价值呢？开关接通后，灯泡过一会儿就会发热；开关断开后，灯泡过一会又会变凉。也就是说，你可以利用热量来判断灯泡是否亮过——哪怕它在你走进房间时是凉的。

于是，你可以像这样来找出开关与灯泡的对应关系：先接通第一个开关，第二

和第三个开关保持断开。十分钟后，断开第一个开关，第二个开关保持不变，接通第三个开关。这样，当你走进房间时，那个不亮但发热的灯泡将对应着第一个开关，不亮也不热的灯泡对应着第二个开关，亮着的灯泡对应着第三个开关。

这道题并没有玩弄文字游戏，也没有什么出格的地方，但确实不容易回答。解答这道例题需要你把思路延伸到题目定义以外的地方。有些面试考官认为这类试题能够帮助他们发现思维“异常”的求职者，即那些不受固定思维束缚、能以创新方式去解决难题的人。但我们认为，这类问题并没有多大价值，也证明不了什么东西。可是，这类问题确实在程序设计面试中出现过，所以大家还是有所准备比较好。

## 8.3 面试例题：过桥

- **有四个人要在深夜通过一座危桥。这座桥最多只能承受两个人的重量，而且必须打着手电筒才能通过。那四个人只有一把手电筒，并且每个人的行走速度也不一样：第一个人通过这座桥要花1分钟的时间，第二个人要花2分钟的时间，第三个人要花5分钟，第四个人要花10分钟；如果两人同行，他们就只能以比较慢的那个人的速度前进。**

**请问：这四个人全部通过这座桥的最短时间是多少？**

因为只有一个手电筒，所以这几个人在过桥之后必须让一个人把手电筒送回来（最后一次除外）。每次过桥的人数只能是一个或者两个；而要想让四个人全部通过这座桥，每次过桥时就必须是两个人，然后让其中的一个带着手电筒返回来。简单计算一下就会知道，总共需要来回5次才能让这四个人全部到达桥的另一端：三次过桥加上两次返回。你的任务是对这四个人进行搭配，让他们全部过桥的时间最短。为了便于讨论，我们将按这四个人的过桥速度给他们编个号，即第1号（过桥用时1分钟）、第2号（2分钟）、第5号（5分钟）、和第10号（10分钟）。

第1号过桥的速度至少要比其他人快两倍，所以让他把手电筒带回来能够把返回时间压缩到最短。这意味着我们应该让第1号一个一个地陪着其他人过桥。

图8-1是这个方案的示意图。第1号陪伴其他人的顺序对总时间没有影响：三趟过桥要分别花费2分钟、5分钟、和10分钟；两趟返回分别花费1分钟、1分钟；所以总时间将是19分钟。

这一方案既有道理又很明显，相信大家都能很快地找出它来。可是，既然这是一道面试考题，这一方案就很可能不是最佳的。也许，面试考官会提醒你说最短时间要短于19分钟；可即使没人提示，你也应该察觉出点什么——这个方案实在是太容易找到了。

你陷入了一种进退两难的境地：你知道自己的答案是错误的，可根据你做出的假设，它又是惟一合理的方案。问题到底出在哪儿呢？你开始怀疑这道题是不是在玩文字游戏：把手电筒扔回来？让第二组打着灯笼？别乱想，这道例题还用不着这

类花招。还存在着一种更为有效的人员搭配方案。因为看起来最符合逻辑的方案是错误的，所以肯定是你做的某个假设有毛病。

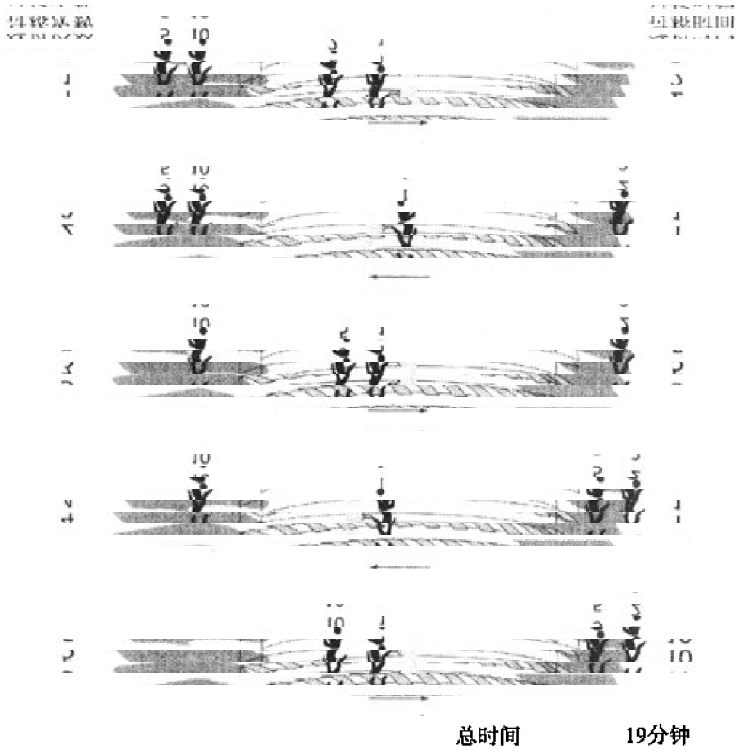



图8-1　由第1号陪同其他人员过桥的方案

我们来分析你做出的假设，看它们哪一个是不成立的。先来看看“过桥和返回必须交替进行”这条假设。它应该是正确的——没有手电筒，那几个人不可能连续两次都是过桥。接下来，“每次过桥是两个人；返回是一个人”。这条假设符合逻辑，但很难证明。让两个人返回等于白忙活，而你的任务是让他们全部到达桥的另一端去；让一个人过桥好像有点道理，但必须再返回一个人，相当于交换了两个人的位置。虽说交换两个人的位置在这道考题里是允许的，但考虑到来回浪费的时间，这多少有点得不偿失的感觉。这条假设看来是做不出什么文章了，我们还是先去分析其他假设比较好。你的下一条假设是“每次都由第1号带着手电筒返回”。你为什么会做出这条假设呢？因为这样做能使返回时间最短。可是，我们的目标并不是让返回时间最短，而是让总时间最短；返回时间最短不见得能保证总时间也最短。“每次都由第1号带着手电筒返回”这条假设有点站不住脚，应该就此做进一步的分析。

如果每次返回的人并不限于第1号，这几个人又该怎样搭配才好呢？你可以用排除

法来解决这一问题。很明显，不能让第10号返回；要不然，他来回共三趟要花费30分钟的时间——就算没有其他人，也要比刚才19分钟的方案差一大截。类似地，让第5号返回也不行：这将导致至少两趟各5分钟的行程，再加上第10号过桥时花费的10分钟，总时间最短也要20分钟才行。总之，让第10或第5号返回都不可能会得出更好的方案。

现在，应该对每次过桥做一下分析了。因为你安排第1号陪同其他人过桥，所以肯定会出现第1号和第10号一同过桥的情况。仔细想想就会明白，让第1号和第10号一同过桥等于是浪费了第1号速度，因为这次过桥还是要花费10分钟的时间。换个角度来看，只要是有第10号参加的过桥行动，不管陪同人员是谁，都必须花费10分钟的时间。既然如此，反正要花费10分钟的时间，为什么不让一个比较慢的人陪同第10号呢？这么一想，让第5号而不是第1号陪同第10号过桥也就有那么点道理了。

可是，如果你安排的第一趟过桥就是第5号和第10号，那么他们中的一个人就必须把手电筒带回来；可根据我们刚才的分析，无论这个带手电筒回来的人是他们当中的谁，都不是最佳方案。你必须在桥的另一端安排一个速度比第5号还要快的人才行。因此，我们把第一趟过桥的人安排为第1号和第2号，然后让第1号把手电筒带回来。现在，桥的另一端已经有一个速度比较快的人（第2号）等在那里了。接下来，我们安排第5号和第10号一同过桥，然后让第2号把手电筒带回来。最后，第1号和第2号再次过桥。整个过程如图8-2所示。

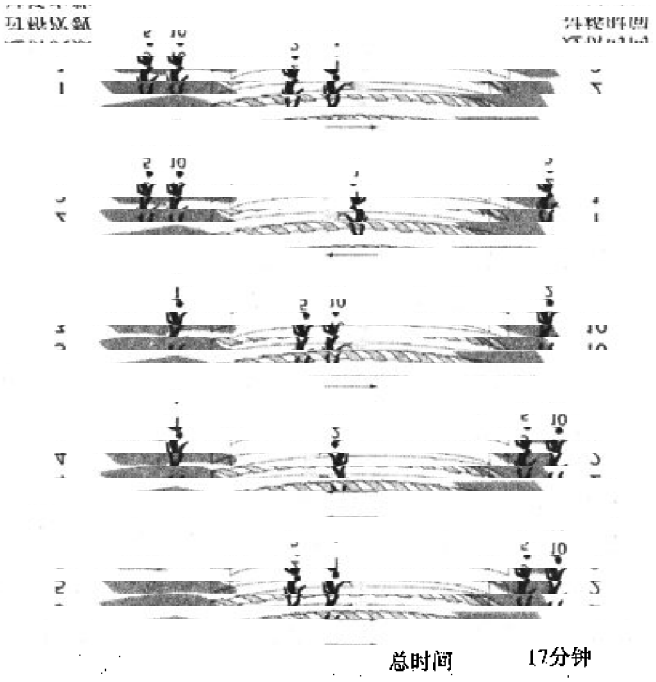



图8-2　按速度进行搭配的方案

采用这一方案的过桥时间依次是2分钟、1分钟、10分钟、2分钟、2分钟，总时间是17分钟。筛选错误假设的思路让我们把总时间缩短了2分钟。

这道例题是“搬运类”问题的典型代表。搬运类考题的基本形式是让你把一些物品从甲地搬运到乙地，要求你花费的总时间最短，或者总搬运次数最少。有时还会增加一些限制性的条件，比如某几样物品必须分在一组，某几样东西不得分在一组，等等。之所以说这道例题有代表性，是因为它还利用了一条错误的假设——“应该让第1号陪同其他人过桥”，这条假设很明显，因为它很容易被想到；同时产生的误导又很隐蔽，因为你往往觉察不出自己已经做了一个这样的假设。

## 8.4 面试例题：找石头

- **给你8颗小石头和一架托盘天平。有7颗石头的重量是一样，另外一颗比其他石头略重；除此之外，这些石头完全没有分别。你不得假设那颗重石头到底比其他的石头重了多少。请问：最少要称量几次，你才能把那颗较重的石头找出来？**

要想正确地解答出这道例题，首先要认识到下面两件事：（1）在进行称量的时候，可以在天平的托盘里放入一颗以上的石头；（2）如果在天平两端放上同样数量的石头，那颗比较重的石头肯定在天平下降端的托盘里。这样，每做一次称量，就可以把天平上升端托盘里的石头都筛除掉，用不着再一颗一颗地去称量这些石头。

看到这儿，很多人会立刻想到要用“二分法”策略来找出那颗重石头。于是，你把石头平均地分成两组并放到天平的两端；然后，筛选掉较轻的那一组，再把较重的那一组平均地分成两组并放到天平的两端……重复以上过程，直到天平两端各有一颗石头为止；如图8-3所示。此时，天平下降端的托盘里的石头就是那颗较重的石头了。这样，你只要进行三次称量就可以把那颗较重的石头找出来了。

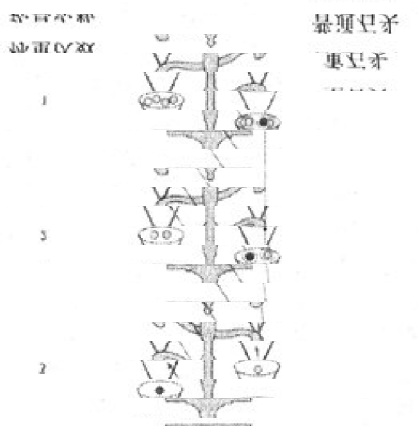


图8-3 用二分法来找出那颗较重的石头

这个答案似乎是正确的：它不那么容易被想到，而且比一颗一颗地称量那些石头的办法要好得多。可如果你认为这个方案还是有点容易，你可就想对了。刚才介绍的办法只是一个良好的开端，不能算做最佳的解决方案。

怎样才能用少于三次的称量找出那颗较重的石头来呢？很明显，你必须在每次称量中筛选掉一半以上的石头才行。可怎样才能做到这一点呢？

请从信息流的角度来看待和分析这道例题。有关这些石头的信息都来自天平，而你则要利用这些信息来寻找那颗较重的石头。从每次称量中获取的信息越多，你的寻找工作就越有效率。再来看看你是如何从天平那里获得信息的：你把石头放在托盘上，再观察天平的动作情况。天平都有哪些动作情况呢？左托盘那端重一些、右托盘那端重一些、两端一样重。也就是说，总共有三种可能的情况——我们刚才介绍的办法只利用了其中的两种。这样看来，你只利用每次称量所能提供的信息的三分之二。如果我们能把每次称量所能提供的信息全都利用起来，说不定就能用更少的称量次数找出那颗较重的石头来了。

在二分法方案里，那颗较重的石头总是要被放在某个托盘里，所以总是会出现天平一端较重的情况。换句话说，只要你把那颗较重的石头放到了某个托盘里，就无法利用天平所能提供的全部信息。那么，如果我们把那些石头分成三组，再用天平来称量其中的两组，会出现什么情况呢？如果天平的某一端较重，那颗较重的石头就肯定在这端的托盘里。如果天平两端重量相同，那颗较重的石头就肯定在没被摆上天平的第三组里。因为你把石头分成了三组，所以每次称量能筛选掉三分之二而不是一半的石头。

在用改进方案来解决这道面试例题之前，还需要再解决一个小问题：怎样把8颗小石头分成三组？因为8不能被3整除，所以你不能把它们均匀地分成三组。为什么要让各组中的石头数一样呢？因为你想让天平两端的石头数相同？仔细想想就会明白，其实你只是需要两组数量相等的石头而已。不过，要想在每次称量中能够筛选掉大约三份之二的石头，还是让各组中的石头数相差不多比较好。

现在，你可以用这个“三分法”方案来解决这道面试例题了。首先，把8颗石头分成三组，其中有两组各有3颗——你将把它们放到天平两端，另一组则有2颗石头。如果天平两端重量相同，那颗较重的石头就在有2颗石头的那一组里——你只需再称量一次就能把它给找出来，即总共需要进行两次称量。如果天平的某一端较重，那颗较重的石头就在这一端的托盘里。在筛选掉其他的石头之后，你再把这一组里的两颗石头分别放在天平的两端：如果天平的某一端较重，你就已经找到了那颗较重的石头；如果天平两端重量相同，没被放上天平的那颗石头就肯定是那颗较重的石头——你总共进行了两次称量。总之，采用“三分法”方案，你只需进行两次称量就能从8颗石头里把那颗较重的石头给找出来；这一过程如图8-4所示。

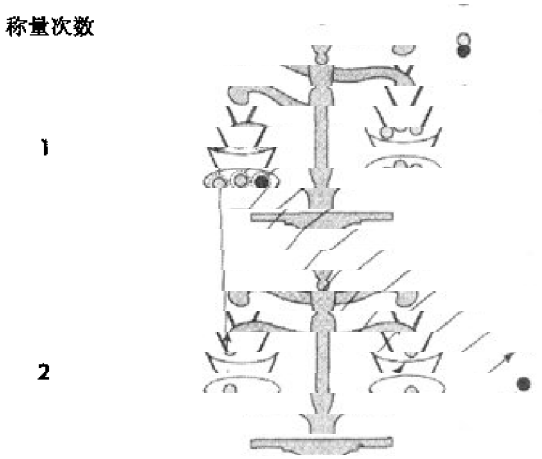



图8-4 最佳方案：用三分法来找出那颗较重的石头

- **请把你的算法推广到普遍情况：最少要称量几次，才能从*n*颗石头里把那颗较重的石头找出来？**

你正确地回答出了上一道例题。是靠运气？还是凭真本领？这道题就是一块试金石。我们来看看对石头进行称量后发生的事情。你筛选掉三分之二的石头，保留了三分之一。每进行一次称量，保留下来的石头就是原来的三分之一。重复这一过程，当每组只剩下一颗石头时，再进行一次称量就能找出那颗较重的石头了。

根据这些描述，你可以把这道面试题重新写为："把石头颗数除以3多少次才能得到1？"如果你有3颗石头，只要除一次3就能得到1，即只需称量一次。如果你有9颗石头，就需要除两次3才能得到1，即需要称量两次。类似地，27颗石头需要进行三次称量。那么，"把石头颗数除以3多少次才能得到1？"的数学表示形式是什么呢？

因为乘法和除法互为逆运算，所以"把石头颗数除以3多少次才能得到1？"与"把1乘以3多少次才能得到石头颗数"这两个问题是等价的。数字自乘在数学里被表示为指数运算。比如说，3自乘两次将被表示为$3^2$，得数是9——从9颗石头里找出那颗较重的石头需要进行两次称量。推而广之，如果从*n*颗石头里找出那颗较重的石头需要进行i次称量，则必然满足等式"$3^i = n$"。数字"*n*"是已知的，你需要把"*i*"计算出来。指数运算的逆运算是对数运算，所以你可以用对数运算求出"*i*"来。对等式"$3^i = n$"的两端做$\log_3$运算，我们得到 $i = \log_3 n$。

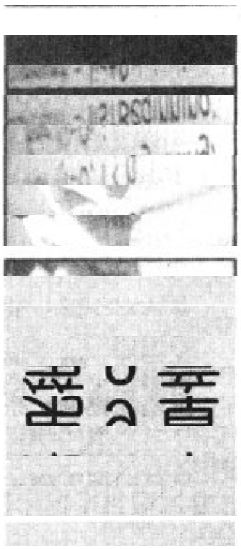
如果*n*是3的某个乘方数，这个公式当然没问题。可如果*n*不是3的乘方数，用这个公式求出来的"*i*"将不是一个整数，而你肯定不能进行非整数次的称量。比如说，如果*n*=8，那么$\log_3 8$将是一个介于1和2之间的实数（准确地说，是1.893…）。如果*n*是3的某个乘方数，这个公式当然没问题。可如果*n*不是3的乘方数，用这个公式求出

来的“$i$”将不是一个整数，而你肯定不能进行非整数次的称量。比如说，如果$n$=8，那么$\log_3 8$将是一个介于1和2之间的实数（准确地说，是1.893…）。根据前面的经验，我们知道8颗石头需要进行两次称量。这样看来，如果计算出来的对数结果是一个实数，我们就应该把它向上舍入为最接近的整数才对。

这么说对不对呢？我们不妨用$n$=10来检验一下。大家知道，$\log_3 9$=2，所以$\log_3 10$应该略大于2。按照刚才的推论，我们应该把它舍入为3。3是在10颗石头里找出那颗较重的石头所需要进行的称量次数吗？当给你10颗石头的时候，你要把它们分成这样三组：其中两组各有3颗石头，还有一组有4颗石头。如果那颗较重的石头在有3颗石头的某一组里，你只需再进行一次称量就可以把它给找出来；如果那颗较重的石头在有4颗石头的那一组里，那你可能还需要再进行两次称量才能把它给找出来（你把4颗石头分为1、1、2三组。如果那颗较重的石头恰好在有2颗石头的那一组里，就需要做两次称量）——与我们计算出来的结果一致。这样看来，计算结果中的小数部分与“找石头”问题的最坏情况（即那颗较重的石头恰好分在数目较多的那一组里）有一定的关系。因为你计算出来的称量次数必须保证你能把那颗较重的石头给找出来，所以你必须把计算结果中的小数部分舍入为一次真正的称量（虽然你不一定需要进行这次称量）。把实数向上舍入为最接近的整数的操作叫做“求顶”（ceiling）。于是，从$n$颗石头里把那颗较重的石头找出来所必需的最小称量次数可以表示为“ceiling($\log_3(n)$)”。

在解答这道例题的时候，不少求职者都能很快地想到并拿出我们最早介绍的“二分法”方案。大多数人都想不到“三分法”，但在经过提示之后，还是能够迅速完成这个解决方案的。这道例题利用了人们的思维定势，故意在已知条件里给出了8颗石头。因为8恰好是2的乘方数，所以很容易把人们误导到“二分法”方案上去；同时，因为8不是3的乘方数（也不是3的倍数），也就很少会有人往“三分法”方案上想了。当这道面试题的已知条件给出的是9颗石头时，找到正确解决方案的人数就会大大增加。像8颗而不是9颗石头之类的细节都是面试考官为了迷惑求职者而故意安排的，如果求职者过分注意这类细节而忽略了题目的内涵，就很容易受到误导而走错方向。

“用天平称量物品”类的题目有很多；公平地讲，“找石头”还算是其中比较容易解答的。如果你想多做一些这方面的练习，请试试下面这道题：“给定一架托盘天平和一些外观完全一样的石头。在这些石头里，有一个与其他的石头重量不同，但不知道它是更轻还是更重。请问：最少需要称量几次，才能把这颗与众不同的石头给找出来。”



# 与图形和空间有关的智力题

许多智力题都与图形或空间有关。适用于非图形类智力题的基本原则也同样适用于这类题目，但在解答这类试题的时候，你又多了一种非常有效的工具：示意图。示意图的作用怎么夸大也不算过分。人类掌握文字和数学知识的时间不过几千年，可解决视觉问题（比如：在我跑到那颗树之前，那头犀牛能追上我吗？）却已经有几百万年的历史了。与用文字或数字来表述的问题相比，人类的大脑更习惯于解决用图形来表述的问题。

**提示：** 只要有可能，就应该用图形来帮助自己。

有时候，图形类智力题里的物体是静止的，但在更多的时候，它们却会不断地变化或改变位置。在遇到这种题目时，不要只画一张图，多画几张是有好处的。根据面试题给出的已知条件，按时间顺序画出有关的动作。根据示意图里的物体移动和变化，你往往能迅速发现问题的本质。

**提示：** 如果问题涉及一系列的变化或动作，请按时间顺序多画一些图来帮助自己。

大多数图形类问题都是二维的。即便涉及到三维物体，它们通常也被限制在一个平面上——你可以把问题简化成二维的。二维图形要比三维图形更容易画，也更容易分析和理解；所以，除非别无选择，你应该尽可能地使用二维图形。如果问题的本质是三维的，那么，在动手之前，你最好先评价一下你自己的三维图形绘制能力和三维思维能力。如果你擅长三维思维但不擅长画图，你画出来的图形说不定反而会帮倒忙。反之，如果你是一位业余画家或绘图高手但三维思维能力较差，绘制一张好的图形就很有帮助了。不管你选择的是哪一条路，图形类问题还是从图形而不

是从数学计算或符号推导来入手更容易解决。

韩坎：三维问题要求你有足够的三维思维能力，但不管怎样，图形类问题还是用图形来解决更容易些。

## 9.1　面试例题：船和码头

- **你坐在一艘小船里，手里握着绳子的一端。绳子的另一端系在离你最近的一根码头桥墩上。你拉紧绳子，让小船朝着码头移动，最终垂直停在了那根桥墩的下面。请问：当你拉动绳子的时候，小船在水面上的移动速度与绳子在你手中拉过的速度哪一个更快？**

看完题目之后，你应该立刻画出一张示意图来，这个图一方面能帮助你更好地理解题目要求，一方面能帮助你开始分析问题。桥墩、水面、绳子构成了一个直角三角形的三条边，如图9-1所示。为了便于讨论，我们将把这个直角三角形的三条边分别称为A、B、C。

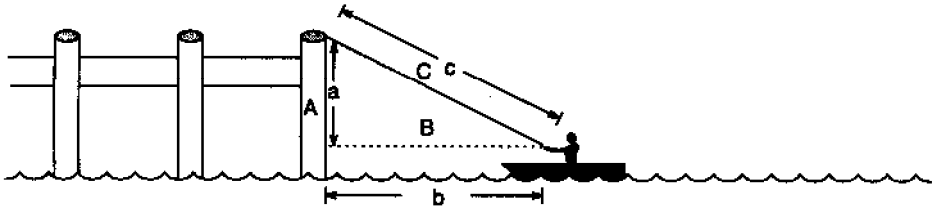



图9-1　水面上的小船

大家对直角三角形应该是很熟悉的了，但这道例题却有一点小变化。你在数学课上肯定遇到过直角三角形，但那都是些静止的图形，而这道例题里的直角三角形却是不断变化着的——它的面积在不断地缩小。请注意这个差异。虽然它不太起眼，却足以让你得出一个看似正确、其实却毫无道理的答案。

根据自己掌握的直角三角形的有关知识，你可能想通过数学计算来解决这个问题。你需要确定直角边B与斜边C在小船移动的时候哪一个缩短的更快。如果换个说法，这道面试题可以表述为：当直角边B缩短给定长度时，斜边C缩短了多少？怎样才能把这些东西计算出来呢？数学中的微积分学能让你求出两个变量的变化速率之比。如果你计算出来的C对B的导数大于1，就能证明绳子移动得更快；反之，如果你求出来的导数小于1，就说明小船移动得更快。

到了这一步，你应该停下来好好想一想了。当然，你可以根据勾股定律建立一个变量为直角边B和斜边C的方程。这个办法应该能够让你得出一个正确的答案。如果你的数学非常之好，并且喜欢求解微积分问题，这说不定还是个最佳的办法呢。

可是，需要用到微积分知识却是一个明显的信号：你可能已经错过了一个更简单的解决这个问题的方法。

你现在应该回到图9-1上来，看能不能通过简单的图形手段来解决这个问题。你还能画出什么图来？因为你既不知道小船与桥墩的初始距离，也不知道桥墩的高度，所以小船运动过程中的那些图无论怎么画，意思都差不多。只有小船停在桥墩下的情况与众不同——不再像图9-1那样是一个直角三角形了，绳子将从桥墩上垂直地挂下来（如图9-2所示）。

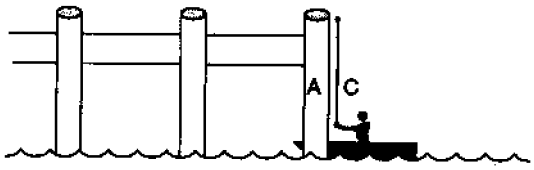



图9-2 小船到达桥墩下

在这前后两个图形里，小船移动了多远的距离？绳子又在你的手中被拉过了多大的距离？因为已知条件里并没有给你具体的数字，所以我们将把直角三角形的A、B、C三个边的初始长度分别用a、b、c来代表。当小船停在桥墩下时，直角边B的长度将等于0，所以小船移动的距离就是b。再来看绳子，它在图9-1里的初始长度是c。在图9-2里，绳子的长度变成了a，所以它被你拉过的距离是（c－a）。因为这两段距离是在同一时间经过的，所以哪一段距离更长，相应的变化速度也就更快。那么，（c－a）和b哪一个更大呢？根据几何学中的“三角形的两边之和大于第三边”（如果三角形的两边之和小于第三边，它们就无法构成三角形了）原理，在图9-1中的直角三角形中，a＋b＞c。不等式两端同时减去a，得到 b＞c － a。即小船经过的距离更长，所以，小船在水面上的移动速度要比绳子在你手中拉过的速度更快。

为了满足好奇心，我们还想再证明一下刚才提到的微积分方案也能得出同样的结论。根据勾股定律，我们先写出公式：$C^2 = A^2 + B^2$。然后，我们再求出C对B的导数，如下所示：

$$C=\sqrt{A^2+B^2}$$

$$\frac{dC}{dB}=\frac{1}{2}(A^2+B^2)^{-\frac{1}{2}}(2B)$$

$$=\frac{B}{\sqrt{A^2+B^2}}$$

B是一个正数，所以当A＝0时，最后一个表达式将等于1。当A大于0时，最后一个表达式里的分母将大于分子，所以表达式肯定要小于1。这就意味着：当B变化了

一个无穷小量时，C的变化量将更小，所以小船的移动速度更快。

这道面试题属于那种“数学知识越多，反而越难解答”的智力怪题。当它们在面试中出现时，“危害性”尤其巨大——因为你本来就准备对付一些很难的题目，并且多少会有些紧张，所以在看到这类题目的时候，就会不由自主地往复杂方面去想，顾不上考虑它是否还有更简便的解决办法。

这类智力题里有一道非常有名的试题，它是这样的：“有两辆汽车都以10公里的时速迎面对开。当它们相距30公里的时候，有一只小鸟开始从一辆汽车飞向另一辆汽车，小鸟的时速是60公里。当小鸟到达另一辆汽车时，它会立刻掉头飞回第一辆汽车。小鸟就这样不停地在两辆汽车之间飞来飞去，直到两辆汽车相撞为止。请问：小鸟总共飞行了多远的距离？”在看完这道题目之后，许多大学生会花上几个小时的时间去建立方程并艰难地计算这个无限序列的总和。而那些还没有学过无限序列的中学生却会这样来解决问题：两辆对开的汽车在相撞之前要经过30公里，它们的时速都是10公里，所以要经过1.5个小时才会相撞。在这段时间里，以时速60公里飞行的小鸟飞过的距离将是$60 \times 1.5 = 90$公里。

## 9.2 面试例题：数方块

- **有一个由小立方体按3×3×3方法构成的大立方体，它的长、宽、高都是3个小立方体｛如图9-3所示｝。**

请问：在这个大立方体的表面有多少块小立方体？

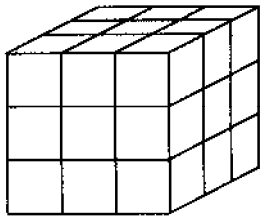


图9-3 一个由3×3×3个小方块构成的立方体

这道面试题需要求职者有足够的空间想像力。空间想像力是一种因人而异的东西，所以我们将在讨论中多介绍几种思路，希望能够对大家有所帮助。首先，你应该画出示意图来，但考虑到这是个涉及到三维空间的问题，你画出来的示意图说不定反而会帮倒忙。

这道面试题最简单、也最没有技巧的解答方法是直接去数大立方体表面的小方块数目。立方体有6个面，每个面有9个小方块（3×3），于是大立方体的表面有6×9=54个小方块。可是，这个大立方体是由总共只有3×3×3=27个小方块构成的，它的表面不可能有54个小方块呀。这个方法错在没有考虑到小方块同时出现在多个面上的情况——比如说，大立方体角部的小方块其实是在3个面上。根据小方块出现在多个面上的情况进行调整是很复杂的，那么还有没有更好的办法呢？

分别数出每一层上的小方块似乎是个更好的办法。大立方体的高度是三个小方

块，所以它总共有三层。最顶上的那一层有9个小方块，它们全都出现在大立方体的表面上。中间一层也有9个小方块，但有一个是藏在大立方体的核心，所以这一层总共有8个小方块将出现大立方体的表面上。底下的那一层有9个小方块，它们也全部出现在大立方体的表面上。于是，9+8+9=26，即大立方体的表面总共有26个小方块。

前面这个办法是行得通的，但数来数去的未免有些麻烦。更好的办法是先把没有出现在大立方体表面的小方块的数目统计出来，然后再用大立方体里的小方块总数减去它。形象生动的图形总是要比虚无的概念更容易理解，所以我们把出现在大立方体表面的小方块全都想像成红色的，把没有出现在大立方体表面的小方块想像成蓝色的。我们希望你能一眼看出这个大立方体其实是一层红色的外壳包围着一个蓝色的核心，而这个核心只有一个小方块。也就是说，没有出现在大立方体表面的小方块只有一个，所以大立方体的表面一定是27 - 1=26个小方块。

- **现在，给你一个由小方块按4×4×4堆放出来大立方体，请问：大立方体的表面有多少个小方块？**

随着小方块数量的增加，一层一层地数小方块的办法就有点捉襟见肘了。我们还是用统计没有出现在大立方体表面的小方块的办法来解决这个问题好了。没有出现在大立方体表面的小方块构成了一个被包含在大立方体中的较小的立方体。这个较小的立方体里又有多少小方块呢？你的第一感也许是认为它只有4块，可4个小方块又怎么能构成一个立方体呢？所以正确的答案是那个较小的立方体是由2×2×2=8个小方块构成的，即没有出现在大立方体表面的小方块总共有8个。大立方体由4×4×4=64个小方块构成，所以出现在大立方体表面的小方块有64 - 8=56个。

- **现在，请把你的解决方案推广到由$n\times n\times n$个小方块堆放出来的大立方体上去。此时，大立方体的表面有多少个小方块？**

现在，数小方块的办法已经根本不能用了，问题也变得越来越有趣。你知道大立方体总共有$n^3$个小方块，如果能把没有出现在它表面的小方块的数量计算出来，也就可以把没有出现在它表面的小方块的数量计算出来了。请发挥一下自己的空间想像力：大立方体的外壳是一层红色的小方块，内部是一个蓝色的立方体核心。它看起来像什么？一个蓝色的立方体被一层红色的外壳包围着，外壳的厚度是一个小方块。蓝色立方体里的小方块数量等于它以小方块为单位的体积。因为蓝色立方体完全被包含在大立方体里，所以它的边长肯定要小于$n$，可它的边长到底是多少呢？

请想像一个穿透大立方体的柱子，这个柱子是由一个一个的小方块摞起来构成的，它的长度是$n$个小方块。因为红色外壳的厚度是一个小方块，所以位于这根柱子首尾两端的两个小方块应该是红色的，柱子中其余的小方块则是蓝色的。也就是说，柱子里有$n-2$个蓝色的小方块。蓝色立方体的长、宽、高是相等的，所以蓝色立方体总共有$(n-2)^3$个小方块。于是，出现在大立方体表面的小方块的数量应该是$n^3-(n-2)^3$个。

我们用前面的例子来检验一下这个公式：$3^3-(3-2)^3=26$；$4^3-(4-2)^3=56$。完全正确！你已经找到了一个具有普遍意义的解决方案，但事情还没有结束。

- **立方体是三维空间中长、宽、高等三维尺寸相等的物体。我们把四维空间中四维尺寸相等的物体称为"超立方体"。请计算出四维尺寸是$n\times n\times n\times n$的4D超立方体表面的小方块数量。**

问题又深入了一层。人类的空间想像力开始遇到了挑战，四维空间里的物体可不是那么容易在脑子里想出来的。虽然困难，但并不是无法可想，大家不妨试试下面的办法。

人们通常把时间称为第四维。把时间想像成实体的办法之一是想像一条电影胶片。胶片上的每一格代表着一个不同的时刻，或者说代表着第四维里的一个位置。为了完整地勾画出四维物体的面貌，你必须把胶片中的每一个想像成一块完整的三维空间而不是二维的图像。如果你能在脑海里勾画出这幅景象，就能想像出四维空间里的物体了。

因为四维超立方体在每个方向上的长度都是相等的，所以你脑海里的胶片也应该有$n$个格长，而每一格里又有一个$n\times n\times n$的三维立方体，这些三维立方体与前面那些例题的立方体没什么两样。这就意味着四维超立方体里总共有$n\times n^3=n^4$个四维方块。再像前面那样给它们涂上颜色，胶片首尾两端的格子是红色的，这两个格子里的三维立方体也全是红色的；中间那些格子则是蓝色的，格子里的三维立方体则与前面几道例题中的一样——蓝色的立方体核心外包围着一层红色的外壳，蓝色的格子有（$n-2$）个。于是，四维超立方体的蓝色核心总共有$(n-2)\times(n-2)^3=(n-2)^4$个四维超方块，而出现在四维超立方体表面上的四维超方块将有$n^4-(n-2)^4$个。

- **把你的解决方案继续推广到$i$维空间：在一个$n\times n\times n\times n\ldots\times n$（$i$维）的超立方体表面有多少个$i$维超方块？**

坚持就是胜利。解决$i$维问题的思路外乎两种：一是发挥想像力去勾画一个$i$维物体；二是撇开想像力直接进行数学推理。我们将用这两种思路来解决这道例题。

一条电影胶片使我们能够想像出一个四维的物体，但我们用不着把思维局限于一条胶片。把$n$条胶片平行地排列在一起就能得到一个五维空间：胶片的每一格是三维，胶片本身又是一维，平行排列着的胶片的序号又能给出一维。在这些平行排列着的胶片中，最左面和最右面的两条是五维超立方体在第五维里的边界，其余胶片则与我们在四维情况里分析的一样。现在来给胶片涂颜色：最左面和最右面的两条胶片里的三维立方体将全都是红色的。如果再加上一个摆放着好几层胶片的架子，就能想像出一个六维的物体来。六维以上的物体想像起来又比较困难了（也许你可以在想像中再增加一些桌子，桌子上摆着一个放有胶片的架子），但我们其实用不着这么辛苦，我们想像出来的东西（胶片、架子、桌子）只是为了更形象地构造出新

增加的那一维而已——多维空间里的物体并不比三维物体更特殊。你每增加一维，就会给你此前想像出来的物体增加$n$个副本。当然，位于第$i$维首尾两端的那两个东西将是i维超立方体在第$i$维里的边界，其余的（$n-2$）个东西则是蓝色的第$i-1$维物体。也就是说，每增加一维，$i$维超立方体里的超方块总数就要比$i-1$维超立方体里的超方块总数增加$n$倍，而没有出现在$i$维超立方体表面的超方块总数就要比没有出现在$i-1$维超立方体表面的超方块总数增加（$n-2$）倍。因此，没有出现在$i$维超立方体表面的超方块总数将是$n^i-(n-2)^i$个。

再从纯数学角度来推导。对于三维物体，我们的计算公式使用了3次方；对于四维物体，我们的计算公式使用了4次方。这样看来，公式里的方次与空间的维数是对应的。如果用一维物体（直线）和二维物体（正方形）来检验这个结论，你将发现它仍然是成立的。因此，从数学角度出发，$i$维空间里边长为$n$个超方块的超立方体应该有$n^i$个超方块；同样的道理，完全被包含在这个$i$维超立方体里的超方块总数应该是$(n-2)^i$个。虽然很难证明，但这已足够让你得出这样一个结论：没有出现在$i$维超立方体表面的超方块总数将是$n^i-(n-2)^i$个。

回顾一下我们得到这一结论的历程是很有意思的。第一道例题相当简单；最后一个问题则近乎不可能——如果没有前面那些步骤的话。例题的难度一点一点地增加，每一次都给你带来了一些新的认识，等你遇到最后一个问题的时候，它已经不像当初那样不可逾越了。请记住这一技巧：困难、复杂、具有普遍意义的问题往往能够通过直白、简单、特殊的问题逐步地得到解决；这往往是一个水到渠成的过程。

## 9.3 面试例题：狐狸与鸭子

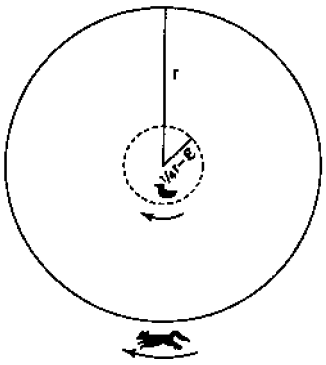
- **一只狐狸在追一只鸭子，鸭子逃到了一个正圆形池塘的圆心位置。狐狸不会游泳，鸭子也不能在水面上起飞（这是一只有残疾的鸭子）。狐狸的速度是鸭子的4倍。假设鸭子和狐狸分别遵循着最优的逃跑和追逐策略，请问：鸭子能不能安全地游到池塘边并起飞？如果能，怎样才能做到？**

鸭子最容易想到的逃跑策略是朝着远离狐狸站立位置的方向直线游进。假设池塘的半径是$r$，则鸭子必须游过的距离就将是$r$。与此同时，狐狸将沿着圆弧跑过圆的半个周长$\pi r$。因为狐狸的速度是鸭子的4倍，而$\pi r<4r$，所以鸭子的这种逃跑策略肯定会让它成为狐狸的美餐。

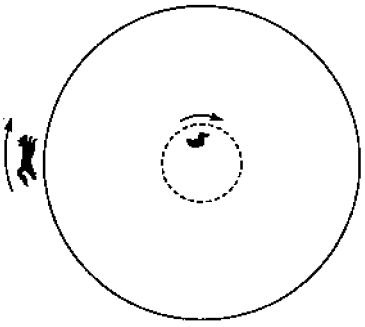
这个结论告诉我们些什么呢？是不是意味着鸭子逃不掉了呢？不是，它只能证明鸭子按这种策略是逃不掉的。要是这道例题真的没有什么“多余的”的东西，那它就只是一个简单的几何练习。但根据题目中的暗示，鸭子是能够逃掉的，只是你现在还不知道它怎样才能逃掉而已。

我们先把鸭子放在一边，去看看狐狸的追逐策略。狐狸必须沿着池塘的边缘追

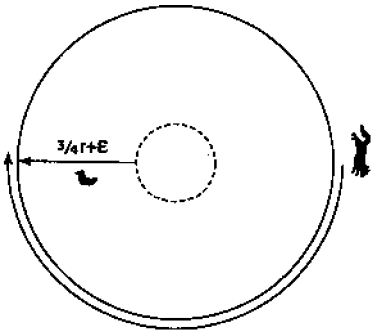
逐并保持与鸭子尽可能近的距离。因为从圆上任何一点到圆心的最短距离都是一条半径，所以狐狸的策略将是尽量与鸭子保持在同一条半径上。




开始逃跑



拉开距离




成功逃走

图9-4　鸭子的逃跑计划

鸭子怎样才能给狐狸造成最大的困难呢？如果鸭子在一条半径上来回地游，狐狸只要坐在这条半径上就行了。可如果鸭子来回地游过圆心，狐狸该怎么办呢？狐

狸将不得不来回地沿着池塘的边缘跑动，反复地从池塘的一端跑到另一端。可是，每当鸭子游过圆心的时候，问题就恢复到最初的状态了：鸭子在池塘的中心，而狐狸在池塘的边上。鸭子的这种策略还是不能奏效。

还有一种可能性：鸭子沿着一个与池塘同心的圆圈游。这样一来，为了保持与鸭子处于同一条半径上，狐狸就不得不沿着池塘边不停地追逐。如果鸭子沿着一个接近池塘边的圆圈游，狐狸就很容易保持与鸭子在同一条半径上——因为它们个自游过和跑过的距离相差无几，而狐狸的速度是鸭子的4倍。但是，如果鸭子的小圆圈越来越收缩于圆心时，这个小圆圈的周长就会越来越短。如果鸭子的圆圈的半径是(1/4) $r$，这个圆圈的周长就将精确地是池塘周长的四分之一；而狐狸则刚好能让自己保持与鸭子在同一条半径上。如果鸭子的圆圈的半径小于(1/4) $r$，狐狸为保持与鸭子在同一条半径上而必须跑过的距离就会超过鸭子游过的距离的4倍。也就是说，当鸭子在半径小于(1/4) $r$的圆圈上转圈的时候，狐狸就要开始落后了。

这个策略似乎能让鸭子在自己于狐狸之间多拉开一些距离。如果鸭子沿圆圈游得距离够长，狐狸所在的半径终将会落后于鸭子所在的半径180°。此时，如果鸭子沿直线游向池塘边，就有可能在狐狸赶到之前到达岸边。怎样才能把狐狸于鸭子之间的距离拉大到如此地步呢？当鸭子的圆圈的半径是(1/4) $r$时，狐狸将刚好保持与鸭子在同一条半径上；当鸭子的圆圈的半径是(1/4) $r$再减去一个无穷小量$\varepsilon$时，鸭子就能稍稍领先。最后，当鸭子领先狐狸180° 时，它与池塘边的最短距离将是(3/4) $r+\varepsilon$。而狐狸必须跑过的距离将是半个周长$\pi r$。此时，狐狸必须跑过的距离将大于鸭子必须游过的距离的4倍（因为$3r<\pi r$）。于是，鸭子将在狐狸赶到之前游到岸边并成功地起飞逃掉。

请大家自行设法去解决一个类似的逃跑与追逐问题："一只狐狸在追一只兔子。它们跑到了一个圆形的体育场里。体育场没有出口，所以它们谁也无法离开。兔子与狐狸的速度是一样的。请问：狐狸能抓住兔子吗？"

## 9.4 面试例题：导火索

- **给你两条导火索和一个打火机。点燃后，两条导火索从一端燃烧到另一端的时间都精确地是一个小时。但它们燃烧的速度并不恒定，彼此也毫无相似之处。换句话说，导火索任意一段的长度与它的燃烧时间没有任何关系，长度相同的两段导火索燃烧时间并不见得相同。现在，请你利用这两条导火索和那个打火机精确地测量出一段45分钟的时间。**

在解答这道面试题的时候，一定要时刻记住这一点：导火索的长度与它的燃烧时间没有任何关系——这道题难就难在这里。虽然这一点已经在题目里反复地得到了说明，但因为人们对导火索"恒定的燃烧速度"、"燃烧长度与时间对应"等非常熟

悉，所以很容易陷入测量导火索长度的误区里。根据已知条件，导火索的燃烧速度是未知且可变的，所以惟一可供利用的测量手段就只有时间了。澄清了这一点，我们才能动手解答这道面试题。

这道题提供给你的材料和允许你采取的动作很有限。在解答这类问题的时候，把所有的可能性都列在一个清单里将对你有很大的帮助；有了清单，事情就要容易的多了。

导火索可以从它的任意一端点燃，也可以从中间的某个位置点燃。无论从哪一端点燃，导火索的燃烧时间都将是60分钟。这比题目让我们测量的时间要长，现在还用不上这一点。如果从中间点燃，就会形成两条火焰，它们将分别燃烧到各自的尽头。如果你的运气够好，恰好在导火索的中点位置（以燃烧时间计，它不一定是导火索的物理中点）点燃了它，两条火焰将在燃烧整整30分钟之后同时熄灭。如果你没有在导火索的中点位置点燃它，两条火焰中就会有一条在不到30分钟的时候熄灭，另一条则在超过30分钟后熄灭。这好像也不能当做精确测量时间的手段。

“在导火索的中间点燃它”与“在导火索的某一端点燃它”得到的时间是不一样的。为什么会这样呢？在半中间点燃会产生两条火焰，相当于同时从两个地方来燃烧这条导火索。这一发现有什么利用价值吗？从半中间点燃导火索好像用处不大，因为你无法确定它们将燃烧多长的时间。那就只剩下从两端去点燃导火索了。如果你同时点燃导火索的两端，两条火焰将越烧越近，直到它们相遇并熄灭为止——时间刚好是30分钟。这一事实应该有用。

现在，你已经能够利用一条导火索准确地得到30分钟的时间了。如果你还能利用另一条导火索准确地测量出15分钟的时间，把这两段时间加起来不就正好是45分钟了吗？用什么东西能测量出15分钟的时间呢？1）一条能燃烧15分钟的导火索，从一端点燃它；2）一条能燃烧30分钟的导火索，从两端同时点燃它。因为导火索的初始长度都是60分钟，所以你必须想法把它去掉45或30分钟。你只能用燃烧的办法来截短它——题目本身已经把其他测量手段都排除掉了。“从两端同时点燃，燃烧22.5分钟”或者“从一端点燃，燃烧45分钟”都可以去掉45分钟；但如果真能做到这一点的话，你岂不是已经把45分钟的时间测量出来了吗？那就只能在30分钟上做文章了。“从两端同时点燃，燃烧15分钟”或者“从一端点燃，燃烧30分钟”都可以去掉30分钟。测量15分钟的时间正是我们此刻在做的事情，而测量30分钟的时间则是我们刚才已经解决了的事情：从两端同时点燃一条导火索，等到火焰熄灭时就得到30分钟了。如果你在同时点燃一条导火索两端的同时点燃另一条导火索的一端，那么，当前一条导火索熄灭时，后一条导火索就必定能继续燃烧30分钟的时间。此时，你立刻点燃后一条导火索的另一端，当它也熄灭时，时间就又过去了刚好15分钟。这样，你就精确地测量出了30 + 15 = 45分钟的时间。

## 9.5 面试例题：躲火车

- **有两位少年从隧道的一端向另一端行走。当他们走过隧道的三分之二时，发现隧道外迎面驶来一辆火车。火车很快就要进入隧道了。两位吓坏了的少年分头向隧道的两端跑去。两位少年的速度都是每小时10公里。两位少年都在千钧一发之际跑出了隧道。假设火车的速度是恒定的，并且两位少年都能在瞬间达到最大速度。请问：火车的速度是多少？**

这好像是中学课本里的一道代数应用题。可是，当你想建立方程的时候就会发现，这道代数题好像缺少了很多已知条件似的：虽然题目给出了少年们的奔跑速度，但却没有给出任何距离和时间。这么看来，这道题要比我们预想的困难得多。

既然没有别的办法可想，那就先根据题目给出的已知条件画几个图试试好了。为了便于讨论，我们把两位少年分别称为A和B。在他们刚发现有火车开来的时候，火车尚在隧道外，距离当然是不知道的了。此时，两位少年的位置相同，都在从距火车较近的隧道口算起的三分之一长度处。我们假设少年A迎着火车跑，少年B向反方向跑，如图9-5所示。

另一个已知道条件是两位少年都安全地躲开了火车。我们来试试能不能画出他们躲开火车时的情景。少年A是迎着火车跑的，只要跑过隧道的三分之一长度就能脱险，所以少年A肯定要比少年B先跑出隧道。因为少年A是在千钧一发之际跑出隧道的，所以在他跑出隧道的同时，火车也进入了隧道。少年B这时候跑到什么地方了呢？少年B与少年A的速度相同，所以少年B在这段时间里也跑过了隧道的三分之一长度。当少年A脱险时，少年B应该在距隧道另一出口三分之一隧道长的地方。此时的情景如图9-6所示。

接下来，我们要画出少年B脱险时的情景。火车通过了隧道，它和少年B应该是同时出现在隧道的另一个出口的（少年A早已跑出了隧道，正暗自庆幸呢）。此时的情景如图9-7所示。

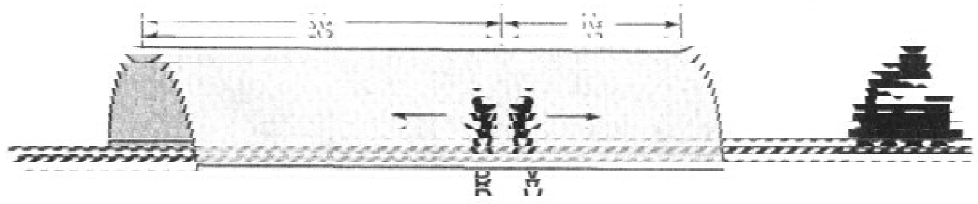


图9-5 少年A和B发现了火车

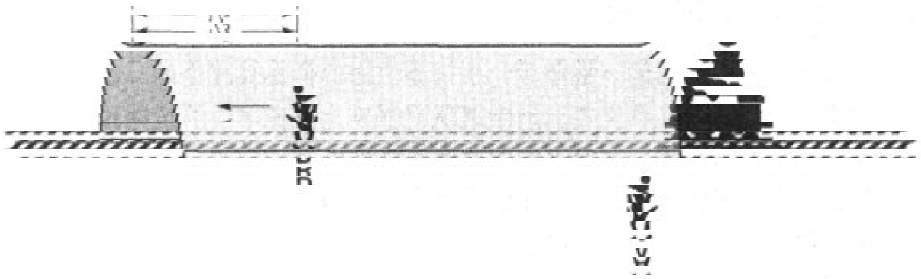


图9-6 少年A跑出了隧道，少年B仍在奔跑

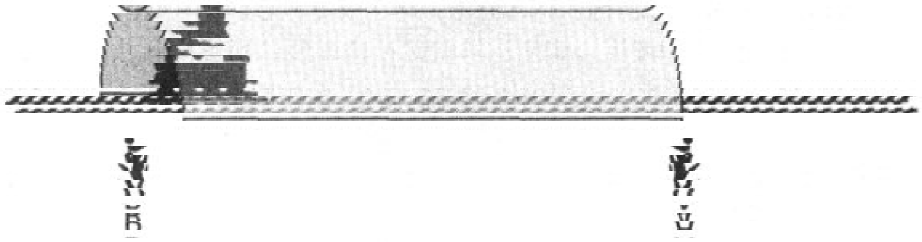
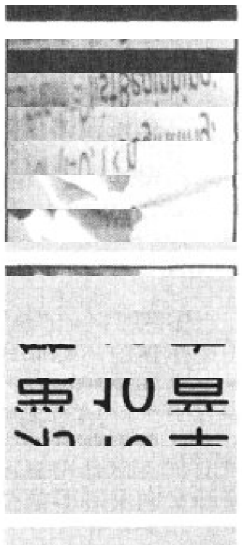


图9-7 少年B也跑出了隧道

如果把这些图割裂开来，它们并不能向我们提供什么线索。我们需要确定火车的速度，所以我们现在来观察一下它的运动情况，看这三个图都能提供哪些线索。从图9-5到图9-6，少年A和少年B各自跑过了隧道的三分之一长度，而火车通过的距离我们无法得知。从图9-6到图9-7，少年B又跑过了隧道的三分之一长度，而火车则通过了整个隧道。也就是说，火车经过的距离是少年B在相同时间内跑过的距离的三倍，这意味着火车的速度也是少年B的三倍。少年B的速度是每小时10公里，所以火车的速度就是每小时30公里。

千万不要低估示意图的威力。



# 计算机基础知识

计算机基础知识考题在不同的程序设计面试会有很大的不同。有的考官可能从不问这类问题，而有的面试考官却可能只问这类问题。面试考官通常会在没有黑板或纸笔的场合——比如与求职者共进午餐时或者在考测完求职者的编程水平后——问这些问题，目的是想进一步了解求职者对计算机知识的掌握程度。根据你的这些问题的回答，面试考官就能判断出你的行业背景和计算机水平。

## 10.1 个人简历

一般说来，计算机基础知识方面的考题主要集中在你的简历上。因此，在参加面试之前，最好能“复习”一下自己的简历并对可能会遇到的有关问题做些准备。有些面试考官会一条一条地根据你的简历发问——“这是什么？”、“你用它做过些什么？”、等等。比如说，如果在你的简历里写有“ActiveX对象”，你就应该对“什么是ActiveX对象？”、“你用ActiveX对象都做过哪些开发？”之类的问题有所准备。如果你最好的回答是：“ActiveX对象能够让我们迅速地开发Web主页。我没有用ActiveX对象做过开发，我只是听说过它们”，那你最好还是把“ActiveX对象”从你的简历里去掉算了。

**提示：** 请对在自己简历上出现过的有关内容加以复习，并做好回答相关问题的准备。

## 10.2 答题要点

就本章而言，要想对求职者简历以及程序设计面试可能会涉及到的各类计算机知识领域做出全面而又系统的论述是不可能的。我们的目标是通过这一章里的例题

让大家对计算机基础知识方面的问题有基本的了解。这类问题主要集中在以下几个方面：计算机的体系结构、各种程序设计技术的优劣对比、程序设计语言的高级功能，等等。这类题材是面试考官最喜欢考的。如果一位号称“计算机字典”的求职者连虚拟内存和磁盘缓冲区等计算机体系结构方面的常用概念都弄不清楚，那他的水平就要大打折扣。另外，很多实际问题并不是以“请用某某语言和某某算法来解决这一问题”的面目出现的，更可能的情况是“我们遇到了这样一个问题，看你能不能解决它”。一位熟知各种解决方案的优劣并知道应该在什么场合使用哪种方案的求职者永远要比那些还搞不清这些东西的求职者受欢迎。最后，这些问题能帮助面试考官评估求职者的工作经验和识别真假千里马。有经验的程序员不可能连他自己惯用的程序设计语言中的高级功能都不了解，而缺乏经验的程序员或者在简历中造了假的求职者却会在这类问题上露出马脚。这些问题能够帮助面试考官识别出求职者中的黄金与黄土。

一般说来，面试考官的提问都比较宽泛，但他们却希望能够得到你比较具体的回答。比如说，假设面试考官问你：“什么是CD-ROM？”比较普通的回答是：“它是一种用来保存数据的东西。”从技术上讲，这个答案并没有什么不对的地方，但却不能证明你真的清楚CD-ROM受欢迎的原因和它们的用途。要是你能口若悬河地说出下面这番话来，效果可就不一样了：“CD-ROM是一种利用光学原理制造出来的移动式数据随机存储设备。与软驱相比，CD-ROM的速度更快，容量更大。真正的CD-ROM是只能读，不能写的。因为价格便宜，所以这种存储介质特别适合用来发行软件。如今，能反复记录和擦写的CD-R和CD-RW正越来越受到欢迎……”

提示：你的回答要尽量做到具体、全面。

注意：本章所给出的例题答案都是人们在事后经过深思熟虑才总结出来的最佳答案。做为求职者，当你在面试过程中第一次听到这些问题时，往往不容易给出如此细致的答案。请把这里的答案当做目标，并尽可能地向它们靠拢。

## 10.3 面试例题：C++和Java

**• C++和Java有什么区别？**

为了吸引更多的C++程序员来学习和使用Java，Java的设计者在Java里保留了很多C++的特点，所以这两种语言在语法上有很多相似之处。除了这些刻意人为的相似之处以外，C++和Java在很多方面各有其特点，这些差异在很大程度上是由这两种语言的设计目的决定的。在设计Java的时候，安全性、可移植性、应用开发速度等因素被放在了最重要的位置，而C++则更注重于性能和与C语言的兼容性。Java程序将被编译为虚拟机的字节码，必须有虚拟机才能运行；而C++程序则会被编译为计算机的

机器代码。所以C++程序的速度往往更快，而Java程序的移植性和安全性则要更高。

C++是C语言的一个超集，继承了C语言的很多特点。比如说，由程序员负责的内存管理机制、指针以及与C语言完全兼容的预处理器（preprocessor）等。Java则取消了这些以及其他一些容易出问题的功能。Java不仅用“废弃内存自动回收”（garbage collection）机制取代了由程序员负责的内存管理机制，还废止了操作符复用（operator overloading）、模板、多方继承等C++功能（Java程序能够用不同的调用接口来有限度地模拟多方继承功能）。这些改进使Java成为一种更有效的开发工具，特别适合用来开发那些可移植性和安全性更重于性能的项目。

在Java里，数据对象都是通过其引用地址来传递的，而C++对象的默认传递方式是值传递。C++会自动进行类型映射，Java却不会。Java里的方法都是虚方法，方法的具体实现是根据对象本身而不是根据其引用指针的类型做出选择的。在C++里，方法必须明确地用被声明为虚方法。Java的基本数据类型的种类是有限的，C++中的类型种类则要取决于它的具体实现。

在需要与旧的C语言程序兼容或者追求高性能的场合，选择C++是有好处的；在强调可移植性、安全性和开发进度的场合，Java将是更好的选择。

## 10.4 面试例题：头文件

**• 预处理器指令 #include "file.h" 和 #include <file.h> 有什么区别?**

这两条指令的主要区别在于编译器查找指定文件的地点。如果使用的是尖括号，编译器将到一系列标准的头文件子目录里去进行查找。如果使用的是引号，编译器将首先查找当前子目录，只有在没能在当前子目录里找到指定文件的情况下，编译器才会到一系列标准的头文件子目录里去进行查找。尖括号通常用于stdio.h等标准的库文件，而引号通常用于程序员自己写的模块。

## 10.5 面试例题：存储类别

**• 请对C语言中的各种存储类别做出解释。**

存储类别（storage class）能够决定变量在内存里的保留时间、指定变量的作用范围、甚至提示编译器进行某些优化。C语言里的存储类别有四种：auto、register、static、extern。

最常用的存储类别是auto。保留字auto只能出现在一个函数的内部。它告诉编译器说这个变量只有在这个函数正在执行时才是必需的；在这个函数前后两次调用之间，用不着保留这个变量的值。用来存储auto类变量的内存通常都分配在堆栈上。在默认的情况下，函数的局部变量全都是auto类的；因此，虽然auto类变量在程序里出现得非常频繁，大多数程序员却很少把保留字auto写出来。

保留字register与auto作用相同，但它还会提示编译器说“这个变量会经常被使用，把这个变量保存在某个寄存器里以减少加载和保存开销将是一个很不错的优化行为”。大多数现代的编译器都会忽略保留字register，因为对编译器进行的测试表明，大多数应用程序员对如何确定最优的寄存器分配方案并不在行。

根据上下文，保留字static有两种不同的含义。在外部层次——即所有函数的外部，这个保留字表示有关变量的作用范围将局限在对它做出定义的那个文件里，不允许从其他文件对它进行引用。在一个函数定义的内部，这个保留字表示应该把有关变量保存到内存中的某个固定地点（而不是堆栈上）；这样，在这个函数的前后两次调用之间，这些变量是值能够被保留下来。

如果你需要在对某个变量做出定义之前就引用该变量，或者需要引用一个在其他文件里定义的变量时，就需要用到extern存储类别。也就是说，保留字extern允许你对变量进行定义，但它并不会创建出这个变量，也不会为它们分配内存。

## 10.6 面试例题：friend类

**• 请对C++中的friend类进行讨论，并举出一个会用到friend类的例子。**

保留字friend（意思是“朋友”）可以作用在函数上，也可以作用在类（class）上。它使“朋友”函数或“朋友”类能够对发生这一定义的类里的私用（private）成员进行访问。有些程序员认为这一功能违背了面向对象的程序设计的基本原理，因为它允许一个类去对另一个类里的私用成员进行操作——如果原来允许其他类访问自己的私用成员的那个类的内部实现发生了变化，它的“朋友”类在访问它的私用成员时就会遇到问题，而这将导致难以预料的bug。

但在某些场合，friend类的使用还是利大于弊的。比如说，假设你实现了一个极其复杂的动态数组类。现在，你想让另外一个类也能遍历你的数组。遍历者类可能需要访问动态数组类里的私用成员才能正确地完成工作。此时，把遍历者类定义为动态数组类的一个friend就很合情合理。这两个类的功能已经有了内在、本质的联系，再教条地把它们割裂开来并没有什么实际的意义。

## 10.7 面试例题：类与结构

**• 在C++里，class与struct有什么区别？**

在不求甚解的程序员看来，这是一个愚蠢的问题。calss（类）允许你拥有成员变量、方法、和继承性，而struct（结构）看起来则要简单的多。事实上，在C++里，class能够做到的事情，struct也同样可以做到。class和struct的本质区别在于访问权限：在缺省情况下，struct里的全体成员都是公用的（public），而class里的全体成员却都是私用的（private）。

为什么C++里会有两个几乎一模一样的构造呢？有三种原因：向下兼容性、设计的自由性、和可移植性。

首先，为了保证与C语言的向下兼容性，C++必须向程序员提供一个struct。既然C语言里的struct已经能把数据组织在一起，进一步增加一些方法和继承性就是很自然的了。第二，C++中的struct定义必须百分之百地保证与C语言中的struct的向下兼容性。C++的设计者之所以要把C++中最基本的对象单元规定为class而不是struct，目的就是为了能够避开各种兼容性要求的限制而自由地给class设置适当的缺省权限。最后，对struct定义的扩展使C语言代码能够更容易地被移植到C++里来。

## 10.8 面试例题：父类与子类

**• 请对C++中的父类与子类之间的关系进行讨论。**

C++里的父类与子类之间的关系主要体现在三个重要方面：继承、虚方法、抽象类中的纯粹虚方法。

子类继承了父类中所有的非私用方法和成员变量。它既可以覆盖继承来的成员，也可以创建新的成员。子类能够被传递为它的父类的一个实例（instance），因为子类里实现有它的父类的全部公用方法和成员变量；但父类却不能被传递为它的子类的一个实例，因为子类可能会实现有一些父类里没有的方法和成员变量。

当子类覆盖一个来自某个父类的方法时，它通常会把这个方法声明为虚方法（virtual method）。如果子类对父类中的某个方法进行了覆盖且把新方法声明为一个虚方法，那么，当以后调用这个方法的时候，系统就会根据调用这个方法的数据对象的类型来选择是执行在父类里定义的方法还是执行在子类里定义的方法；否则，系统就会根据用来调用这个方法的引用指针的类型做出选择。

比如说，请看下面的父类和子类定义：

```
public class Lion {
    public void Roar() {
        PlaySound(EARTH_SHAKING_ROAR);
    }
}

// subclass of Lion
public class BabyLion : Lion {
    public void Roar() {
        PlaySound(LITTLE_MEOW);
    }
}

// Function that calls Roar
void MakeMovieStudioSound(Lion *leo) {
```

```
    leo->Roar();
}
```

很明显，BabyLion类与Lion类是父子关系，它们都能被传递到函数MakeMovieStudioSound里去。现在，把一个指向BabyLion的指针被传递到这个函数里去。如果Roar不是一个虚方法（就像上面那段代码中一样），计算机就会执行在Lion里定义的Roar实现而不是执行在BabyLion里定义的Roar。系统也不进行任何实时检查以分辨被调用的到底是Roar的哪一个实现。

如果Roar是一个虚方法，计算机就会实时检查数据对象的类型，进而发现该对象其实是一个BabyLion，于是就会去调用在BabyLion里实现的Roar。

这是一个很有意思例子。不过，非得用非虚方法才能完成事情并不多，所以程序员也很少使用这种手段。但调用虚方法花费的时间要比调用非虚方法的时间多一点，所以在缺省的情况下，C++中的对象方法都是虚方法。

父类-子类关系的第三个方面与纯粹虚方法和抽象类有关。纯粹虚方法（pure virtual method）指的是在父类中没有定义但被子类继承的方法。带有纯粹虚方法的类就叫做抽象类（abstract class），抽象类是不能被实例化的。如果父类是一个抽象类，那么它的子类要么把所有的纯粹虚方法都覆盖掉，要么它也成为一个抽象类。比如说，假设父类Clown里有一个名为Juggle的纯粹虚方法。那么，当你创建它的子类SadClown时，就继承了这个纯粹虚方法。如果SadClown没有用一个定义覆盖掉Juggle，那么SadClown就将成为另一个抽象类。

## 10.9 面试例题：参数传递

- **下面是C++函数foo的几种调用模型，请问：它们中的哪个是以Fruit类的对象为输入参数的?**

```
void foo(Fruit bar);         // Prototype 1
void foo(Fruit* bar);        // Prototype 2
void foo(Fruit& bar);        // Prototype 3
void foo(const Fruit* bar); // Prototype 4
void foo(Fruit*& bar);       // Prototype 5
```

**请说出各调用模型的输入参数是怎样传递的，不同调用模型的函数又应该怎样去实现?**

在第一种调用模型里，对象参数是按值传递的，系统将调用Fruit类的构造器的一个副本把有关对象复制到堆栈上去。在这个函数里，bar是Fruit类的一个对象。因为bar是被传递到函数去的对象的一个副本，对bar的修改将不会反映在原来的对象里。这种调用办法的效率是最低的，因为对象的每一个数据成员都必须进行拷贝。

在按第二种调用模型实现的函数里，bar是一个指向一个Fruit对象的指针。与按

值传递的办法相比，因为只需把对象的地址而不是对象复制到堆栈上去，所以这种办法的效率要高很多。因为bar是被传递到函数中的对象的指针，所以通过bar进行的修改都将反映在原来的对象里。

第三种调用模型里的bar是按引用地址传递的（passed by reference）。它与第二种调用模型非常相似：不需要复制对象且允许foo直接在父函数的对象上进行操作。第二种与第三种调用模型的主要区别在于它们的语法。如果传递的是指针，在访问成员变量和函数前就必须对指针进行退指针；如果传递的是引用地址，不需要做类似处理就能访问有关成员。因此，如果输入参数是个指针，就需要用箭头操作符（->）来对成员进行访问（你也可以用“*”加“.”操作符达到办法来进行退指针处理。因为“.”比“*”操作符的优先级高，所以你必须给它们加上括号：(*bar).property = 1；）；如果采用按引用地址传递机制，就需要使用“.”操作符。一个容易被忽略但更为重要的区别是：指针不必非得指向一个合法的Fruit；foo函数的指针输入参数可以是一个NULL指针。但在第三种调用模型中，bar必须是一个合法的Fruit的引用地址。

在第四种调用模型里，bar将被传递为一个指向Fruit对象的常数指针。这种办法具有第二种调用模型的高效率，但函数foo将不能对bar进行修改。只有被定义为const的方法才能从函数foo里对bar进行调用——这是为了防止foo对bar做间接修改。

在最后一种调用模型里，bar是一个指向Fruit对象的指针的引用地址。类似于第二种调用模型，对这个对象做的修改也能够在父函数里“看”到。此外，因为bar是一个指针的引用地址而不仅仅是一个指针，所以，如果我们让bar指向另外一个Fruit对象，从父函数传递过来的指针就也将被修改。

## 10.10 面试例题：宏与内嵌函数

**• 请对C++中的宏和内嵌函数进行对比。**

当预处理器遇到程序代码里的宏（macro）时，它将简单地进行文本替换。比如说，如果你定义了一个如下所示的宏：

```
#define AVERAGE(a, b) ((a + b) / 2)
```

那么，预处理器就会把出现代码中所有的AVERAGE(a, b)替换为( (a + b ) / 2 )。如果你在代码里会频繁地用到一些很难书写或者记忆的东西，但又简单得不值得把它们写成一个函数，就可以为它们定义一个名字简单易记的宏。

内嵌函数（inline function）的声明和定义办法与普通函数的非常相似。与宏不同的是，它们将直接由编译器来处理。用内嵌函数来实现的AVERAGE宏如下所示。（如果把成员函数（即方法）的定义放在类的定义里，就可以把它们隐含地声明为内嵌函数。注意：必须对内嵌函数的输入参数和返回值类型做出声明；宏则没有这方

面的要求。利用模板可以在类型定义方面获得一定灵活性，但这与这道面试例题的范围离得太远了。）

```
inline int Average(int a, int b)
{
    return (a + b)/2;
}
```

站在程序员的立场看，调用一个内嵌函数与调用一个普通函数没什么两样。但在编译器的眼里，它们还是有区别的。如果编译器遇到的是一个内嵌函数，它将把一份经过编译的函数定义副本插入到相应的位置上，而不是生成一个函数调用。

内嵌函数和宏是以增加程序长度为代价来减少函数调用开销的两种手段。内嵌函数与函数调用有相似的语法，而宏使用的则正是文本替换的语法，这就有可能因为宏的行为出轨而产生bug。

请看下面的宏定义和代码：

```
#define CUBE(x) x * x * x

int foo, bar = 2;
foo = CUBE(++bar);
```

你本以为这段代码会把bar设置为3，把foo设置为27，但实际生成的代码却是下面这样的：

```
foo = ++bar * ++bar * ++bar;
```

因此，bar将被设置为5，而foo将被设置为60。如果你把CUBE实现为内嵌函数，就不会出现这样的问题了，类似于普通函数，内嵌函数只对其输入参数进行一次求值，所以因求值而产生的副作用只会发生一次。

下面是在宏的使用过程中容易出现的另一个问题。假设你定义了一个有两条语句的宏，如下所示：

```
#define INCREMENT_BOTH(x, y) x++; y++
```

如果你在if语句的语句体只包含一条语句时习惯于不使用花括号，就可能会在程序里写出这样的代码：

```
if (flag)
    INCREMENT_BOTH(foo, bar);
```

你原以为这段代码将被扩展为：

```
if (flag) {
    foo++;
    bar++;
}
```

可惜，当这个宏被扩展时，if将只绑定宏定义里的第一条语句，于是就得到了下面这样的代码：

```
if (flag) {
    foo++;
}
bar++;
```

内嵌函数就不同了：不管本身包含有多少条语句，它在程序代码里只是一条语句，所以内嵌函数不会出现上面这样的问题。

避免使用宏的最后一条理由是：在源代码里看不到将要被编译的、由宏扩展而来的代码。这使得与宏有关的bug非常难于查找和调试。一般说来，应该尽量使用内嵌函数，避免使用宏。

## 10.11 面试例题：继承

**• 已知如图10-1所示的类的继承关系，并给你一个以B*为输入参数的方法。请问：图10-1给出的类里哪些能被传递到这个方法里去？**

首先，根据已知条件，可以把B传递到这个方法里。其次，D应该不能被传递到该方法里去，因为D极可能与B有完全不同的构造。A是B的父类；我们知道，子类必须实现父类中的全部方法，而父类却不必包含子类中的所有方法，所以我们不把父类A传递到该方法里去。C是B的子类，必须实现B中的全部方法，所以我们可以把C迟到该方法里去。

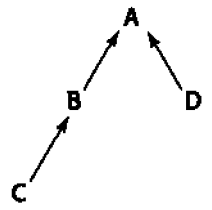



图10-1 类的继承关系

## 10.12 面试例题：面向对象的程序设计

**• 与非面向对象的程序设计技术相比，面向对象的程序设计技术到底好在哪里？**

面向对象的程序设计技术（object-oriented programming，OOP）最明显的特点是把有关数据以及对这些数据进行操作的方法关联在了一起，使人们能够更迅速地为现实世界中的事物建立起模型来。此外，OOP技术中的类（class）不允许从外部直接对它的成员变量进行访问，而成员变量的值在数据对象的生存期内一直能够得到保持。如果某个类以外的其他程序零件想访问这个类里的成员变量，就必须通过这个类提供的方法来完成有关的数据检索和数据修改操作。这使程序员更容易把整个程序划分为一个一个的功能模块，并把有关的数据对象的实现细节全部隐藏在类的内部，从而使大型项目更容易管理和维护。

不过，这些特点也能通过非面向对象的程序设计技术中的模块或类似构造实现

出来。比如说，C语言提供有一种大家都比较熟悉的构造叫做“抽象数据类型”（abstract data type，ADT）。这是一些能够把有关数据和对这些数据进行操作的函数关联在一起并允许外部调用者对有关数据进行检索和修改的模块。这种关联是通过把一个ADT定义为一个指向某个struct结构的指针的办法而实现的。这个struct结构里的值就相当于OOP里的成员变量，需要访问这些值的函数必须把这个ADT做为输入参数之一。这就可以把数据和函数绑定在一起了。与C++或Java中的使用的类相比，这种做法是显得不够精巧，但确实能够使程序具备同样的特点。

OOP真正与众不同的地方是它的继承（inheritance）机制。用非面向对象的程序设计语言来实现继承机制几乎是不可能的。继承是一个非常重要的概念，它使大型函数库更容易管理和维护，也使代码的共享和再利用更加方便。自从有程序设计以来，人们已经通过函数库积累了大量的优秀代码，而OOP的继承机制则使这些优秀代码（或者叫函数库）的使用更加简便。继承机制使程序员能够在不改变库函数代码的前提下根据自己的需要对函数库进行修改或裁剪，使程序设计工作更容易进行。同时，继承机制还使程序员能够利用继承自基类（base class）的函数去完成各种相似或者重复性的工作。这种代码的再利用大大加快了软件项目的开发进度。继承机制使OOP成为了一种更好的程序设计技术。

## 10.13　面试例题：与线程有关的程序设计问题

**• 请对线程程序设计工作中的常见失误进行分析。**

线程程序设计工作中的常见失误有竞争（race condition）、死锁（deadlock）、活锁（livelock）、忙等待（busy waiting）、和锁死（over-locking）。

如果没有对线程进行正确的同步，就会产生竞争现象。竞争通常发生在两个线程同时访问并修改同一个全局变量的场合。请看下面这个例子：两个线程都要对某个全局变量做递增操作，假设这个全局变量的初始值是5。

递增操作通常需要三条机器代码才能完成：一条用于加载、一条用于修改、一条用于保存。首先，第一个线程进行了加载，得到数值5，然后就被切换出去了。接着，第二个线程也进行了加载，得到数值5，给它加上1，然后把它保存为数值6。再往后，控制又返回到第一个线程，它此时仍认为那个变量的值是5；第一个线程将进行同样的加法操作并把数值保存为6。此时，全局变量做了两次加法，但结果却是不正确的6而非正确的7，造成这一现象的原因就是不正确的线程同步。

死锁是指这样一种现象：两个（或者多个）线程都在等待对方拥有着的数据锁，因而谁也不能继续执行。比如说，线程A拥有着一个数据锁，但必须等待线程B拥有着的数据锁被释放后才能继续执行，而线程B却必须等待线程A拥有着的数据锁被释放后才能继续执行。于是，两个线程形成了死锁的局面，谁都不能继续执行。这种

情况极其严重，有关程序将因为死锁而陷入停顿。

活锁是因死锁而导致的一种情况。有些机器具备检测死锁现象的机制。当检测到死锁现象时，系统将让有关线程放弃它们各自拥有着的数据锁并继续推进。但是，在所有的数据锁都被释放之后，这些线程却有可能会经过同样的程序代码而再次相遇并导致与刚才一样的死锁局面。人们把这种“死锁、放弃数据锁、再次死锁“的过程称为活锁。此时，虽然有关线程并没有处于死锁状态，却都无法完成它们各自的任务。

忙等待也是线程程序设计工作中的常见失误之一，虽说不会影响到程序的正常工作，但它对系统的性能却有很大的不利影响。忙等待是指这样一种局面：某线程苏醒过来，发现自己必须等待某个资源才能继续执行，于是就不停地检测该资源是否已被释放，直到该线程再次进入休眠等待状态。这个问题的症结在于：虽然线程没有停滞，但除了不停地检测同一资源外却不能做任何有用的事情。人们使用线程的目的本来是为了提高有关程序的执行效率，但因为出现忙等待现象的缘故，不仅有关程序自身的性能下降了，还无谓地消耗了系统的资源。消除忙等待的办法是：在有关资源变得可用之前，应该让有关线程一直休眠下去，直到该资源变得可用为止。

锁死是另外一种会对系统性能造成不利影响的现象。请看下面这段代码：

```
wait(semaphore);
/* value is a global variable that needs to have its access
 * synchronized.
 */
value++;

/* lots of time consuming stuff that doesn't have to be
 * synchronized
 */

/* ... */

signal(semaphore);
```

大家看出什么问题来了吗？虽然这段代码本身的执行没有问题，却不必要地把有关资源“霸占“了太长的时间：在它执行大量消耗时间的代码的同时，其他需要这个资源的代码却只能够等待。锁死现象会引发严重的性能问题，把使用线程的好处抵消得一干二净。

## 10.14 面试例题：废弃内存的自动回收

- **请对“废弃内存自动回收”（garbage collection）机制及该机制在Perl和Java中的实现情况进行讨论。**

“废弃内存自动回收”中文直译的意思是“垃圾回收”）机制是一种由程序自动

释放已分配内存的机制，这一机制不需要程序员的参与和帮助。Java、Lisp、和Perl是实现有“废弃内存自动回收”机制的程序设计语言的代表。

与由程序员负责释放内存的“低级”做法相比，“废弃内存自动回收”机制有很多优点。首先，它能彻底消除因悬挂指针（dangling pointer）和内存泄漏（memory leak）等常见问题而导致的程序漏洞。其次，它大大简化了程序和程序接口的设计工作——在编写代码的时候，程序员不必再为如何添加适当的机制才能保证内存能够被正确地释放而伤脑筋了。因为程序员不再有释放内存的负担和压力，所以使用具备“废弃内存自动回收”机制的程序设计语言来开发软件往往要比使用只有传统内存管理机制的程序设计语言来得更快。

但“废弃内存自动回收”机制也有它不足的地方，效率不高就是它的弱点之一。因为系统必须花时间去判断哪些内存需要被回收并进行这种回收，所以使用了“废弃内存自动回收”机制的程序往往运行得比较慢。此外，为了保证不影响程序的运行，由系统分配的内存往往会超出程序实际需要的数量，也不会在第一时间把它们回收回来。因此，人们一般都不会使用具备“废弃内存自动回收”机制的程序设计语言来开发那些看着重于速度而看轻内存开销的应用程序。

Perl与Java中的“废弃内存自动回收”机制有着很大的不同。Perl使用的是一种名为“引用计数器”的机制。这一机制的工作原理是：为数据对象分配必要的内存并把有多少个变量引用了这个对象的情况记录下来。起初，某块内存的引用计数器的值是1。此后，每增加一个引用这块内存的变量，这个引用计数器就会加上1；每当有一个变量改变了取值或者离开其作用范围时，这个引用计数器就会减去1；当这个引用计数器的值变成0时，Perl就将释放这块内存。此后，因为你已无法再访问这块内存，这块内存就将被系统回收。

“引用计数器”机制的优点是原理简单，速度较快；它的不足是无法处理“循环引用”情况。举个例子：假设有一个循环链表，但没有外部变量引用它。此时，链表中每个元素的引用计数器的值都是1，但整块内存却没有引用计数值。因此，整块内存应该被回收才对，但Perl的“废弃内存自动回收”机制却无法释放它。Perl程序员要特别注意这一点：在释放循环性数据结构的最后一个外部引用之前，一定把循环性引用的“怪圈”打破才行。

从这一点上讲，Java的“废弃内存自动回收”机制要比Perl的完善的多，程序员不再需要警惕这类特殊的情况。Java的策略是“先标记，再清除”：即在某一时刻，内存管理器先给所有正在使用的内存单元加上一个标记；然后，在第二遍扫描里，把所有没有标记的内存单元清除掉——即加以释放和回收。

这种做法的好处是不再有需要程序员特别警惕的情况（例如Perl中的循环性链接结构），但效率却比较差。此外，因为标记和清除动作不一定发生在程序执行过程中

的同一位置，所以内存的释放情况是不可预料的。

## 10.15 面试例题：32位操作系统

**• Windows NT是一个32位的操作系统。可32位的操作系统到底是什么意思呢？**

首先，一个32位操作系统至少要运行在一个32位的处理器上；其次，它通常还必须具备向程序提供32位虚拟地址空间的能力。运行在32位操作系统上的程序通常使用32位做为最小字长。比如说，整数将是一个32位而不是16位的值。此外，“32位操作系统”这个词通常还隐含地表明这个操作系统使用了一些先进的技术，如多用户多任务技术、进程孤立技术等，但这些技术并不是32位操作系统之所以成为32位操作系统的必要条件。

## 10.16 面试例题：网络性能

**• 评判网络性能的两大标准是什么？**

任何一个网络都可以用两个指标来衡量：传输延迟（latency）和带宽（bandwidth）。传输延迟指的是一个给定的信息比特穿越网络所花费的时间；带宽则是数据在通信链路建立后在网络上的传播速度。拥有无限带宽和零传输延迟的网络是最完美的。

如果把网络比做一根水管，那么，水分子流过水管的时间就相当于网络的传输延迟，在给定时刻流过水管截面的流量就相当于网络的带宽。

传输延迟和带宽问题可以利用浏览器来判断：当你浏览Web的时候，如果页面等了很长时间才开始显示，但立刻就全部出现了，就说明网络的传输延迟比较大，带宽还不错。如果页面立刻开始出现，但用了很长时间才全部显示出来，就说明网络的传输延迟还可以，但带宽却偏小。

## 10.17 面试例题：高速磁盘缓存

**• 你写的一段代码需要到硬盘上去检索数据，但频繁的硬盘访问却降低了整个系统的性能。现在，你的老板让你设法对此做些改进，但是要求：1）这些信息检索操作不能省略或者绕过；2）不得更换高速硬盘、高速总线、或高性能的文件系统。请问：用什么办法才能解决这一问题？**

你的老板太坏了！这与让你把一辆最高时速为60公里的汽车开到80公里没什么两样。不过，现在还不是着急发火的时候，把面试考官的真正用意弄清楚才是最重要的。因为没有给你任何细节，所以你的回答也不必纠缠于细节。面试考官感兴趣的是你能不能找出具体问题的原理性解决方案。根据这一思路，我们把这个问题重新表述为：“提高硬盘访问速度的原理性办法是什么？”

这个问题就简单了。根据例题中的限制条件，提高硬盘访问速度的原理性办法就只剩下“磁盘缓存技术”了。简单地说，磁盘缓存技术将把硬盘上最近被访问过的一些数据块保存在主内存里；在对硬盘进行读写之前，计算机先要检查有关数据块是否已经在缓冲区里了。如果数据块已经在内存里，就从内存读出信息——这比读硬盘要快多了；如果不在，计算机再从硬盘上把它读出来——为提高效率，计算机会把与指定数据块相邻的几个数据块也同时读入缓冲区。与硬盘相比，内存缓冲区里的数据块的数量是很少的，但因为程序往往会重复访问同一批数据，所以磁盘缓存技术对磁盘的性能也往往会有很大的提高。

符合你老板要求的答案只能是使用磁盘缓存技术了；这当然不是最佳的解决方案。一般说来，磁盘缓存技术通常用在比较靠下的OS层而不是比较靠上的应用层，在应用层使用磁盘缓存技术很容易产生各种各样的bug。比如说，当其他用户的程序准备对被你在应用层缓存了的硬盘数据块做写操作时，就必须设法把这一事件通知给你的程序，否则，缓冲区里的信息就将不再是最新的了。另外，这道例题并没有把硬盘数据的访问模式告诉你，所以磁盘缓存技术能否缓解你遇到的问题还是一个未知数。不过，在题目给定的限制条件下，磁盘缓存技术将是惟一可能的解决方案。

## 10.18 面试例题：数据库的优点

**• 与你自己去实现一个数据存储子系统相比，把信息保存到数据库里有什么好处？又有什么坏处？**

使用数据库的好处是1）便于对有关功能进行扩展；2）提高数据的可管理性；3）减少bug。首先，数据库能够提供多种数据分析，数据备份，数据查询手段。其次，与你自行实现的数据存储子系统相比，数据库更容易管理和维护。从理论上讲，只要懂得数据库的使用方法，任何人都能接过你已开发的代码并继续进行开发。再其次，数据库系统通常都经过严格的测试，程序漏洞相对要少得多。

说到不利因素，首先，数据库通常都比较昂贵（当然，也有一些数据库系统是免费的）。其次，一旦你开始使用某种数据库产品，往往就只能继续依赖同一家供货商了。还有，信息在数据库里的进进出出往往要产生巨大的开销，如果你的程序不需要做大量的搜索、索引、关联等操作，就没必要以牺牲系统性能为代价去追求数据库。此外，如果你的项目比较小，用数据库来做为它的存储子系统就未免有点大材小用或者喧宾夺主的味道。

## 10.19 面试例题：加密技术

**• 请举例说明对称密钥加密技术和公开密钥加密技术之间的区别。**

对称密钥加密技术也叫共享密钥加密技术，对信息进行加密和解密时使用的密

钥是完全相同的。公开密钥加密技术则使用了两把不同的密钥：一把公开密钥，用来对信息进行加密；一把私用密钥，用来对信息进行解密。与公开密钥加密技术相比，对称密钥算法的优点是：速度快、容易实现、专利或知识产权纠纷少、对硬件的处理能力要求不高；它的缺点是：在正式的加密通信开始之前，双方必须先就密钥达成一致。这不仅很不方便，在某些情况下甚至是不可能的。如果通信双方位于不同的地点，首先就要有一个保密的途径来传递密钥。在只允许使用对称密钥的场合，绝对保密的途径往往是不存在的——如果真有一条如此保密的途径的话，双方也就用不着再建立一条保密的通信通道了。

公开密钥加密技术的优势在于不需要把加了密的信息隐藏起来就能保证它的安全。也就是说，公开密钥可以通过不安全的途径进行传送。因此，人们经常先通过公开密钥加密技术来确立一个对称密钥，再用这个对称密钥通过对称密钥加密技术进行通信。这样，既能享受公开密钥加密技术带来的方便，又能获得对称密钥加密技术的高性能。

在Web上，这两种加密技术都被用来获取需要保密的信息。首先，你的浏览器通过公开密钥加密技术与远方的Web站点确立一个对称密钥，然后再用这个对称密钥通过对称密钥加密技术与Web站点交换需要保密的信息。

## 10.20 面试例题：新的加密算法

**如果你发现了一个新的加密算法，你会立刻使用它吗？**

这个问题本身并不难回答，但它却是现代加密技术的一个敏感而又核心的问题。原则上讲，没有一种算法能够永远不被攻破，即使算法本身没有漏洞，用来实现这种算法的程序代码也多少会有几个bug。加密算法是由人研制出来的，是供人们使用的，而现代高级黑客所使用的技术也是顶尖的，这些因素使得任何一种加密算法迟早都会被别人攻破。如果你的加密算法要靠不让别人知道才能确保无虞，即“以不知换安全”，那其实就毫无安全可言；它迟早会让一个有恒心的黑客攻破。更为严重的是，它会让你有一种虚假的安全感：你自认为是安全的，可实际上却早已被别人给“暗算”了。

因此，你应该从一开始就把自己的加密算法公开出来，让大家帮你清除漏洞；如果一直藏而不露，等发现它存在着严重的安全隐患时就追悔莫及了。当然，密钥还是不让别人知道的好。在经过公众广泛而又深入的研究和讨论后，只有新算法的安全性得到了大家的承认时，你才能够把它投入实际应用。

## 10.21 面试例题：哈希表与二元搜索树

- **请对哈希表和二元搜索树进行对比。如果让你为内存有限的手持设备——比如Palm Pilot——的地址簿设计一种数据结构，你会选用哪一种？**

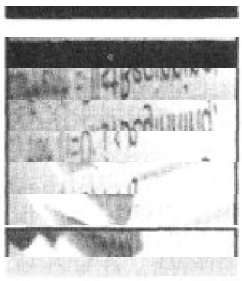
哈希表只擅长做一件事：能够非常快速地存储和检索数据（执行时间是一个常

数O(1)），但其他事情就力不从心了。

二元搜索树的插入和检索时间是O(log($n$))级的——虽说比不上哈希表的O(1)，速度也算是快的了。除此之外，二元搜索树还有一个特点：它的元素是排好序的。

对手持类设备来说，你应该尽可能多地把它的内存留做数据存储之用。如果你使用的是哈希表等无序的数据结构，就需要额外的内存来对有关数据进行排序——因为你肯定需要按字母表或者其他顺序来显示数据。也就是说，如果你使用哈希表的话，就必然会挤占一些宝贵的内存。

如果你使用的是二元搜索树，就不必浪费内存或处理时间去对数据进行排序了。虽说二元搜索树上的操作比哈希表慢，但手持类设备最多也不会有10 000条以上的数据条目，所以二元搜索树O(log($n$))级的执行时间无疑是足够快的了。因此，对于这类任务，二元搜索树将比哈希表更适用。



# 非技术问题

大多数技术类面试都会有一些非技术性的问题。这些问题一般会出现在整个面试过程的初期阶段，目的是为了确定你的工作经验和求职目标，看你能不能胜任有关职位。很多程序设计面试考官也会问一些非技术性的问题，因为你们很可能会每星期共同工作40个小时。

在传统的面试里，所有问题都是非技术性的，面试官只能通过你对这些问题的回答来了解你。在技术类面试里，你必须在技术问题上有突出表现才有可能获得工作邀约——没有人会仅凭你在非技术性问题上的优异表现而给你一份工作。但是，如果你在非技术性问题上表现不佳的话，说不定煮熟的鸭子就会飞掉。

非技术问题看起来都比较简单，尤其是当你已经在艰难的技术类测试中杀出重围的时候。但从找工作的大局上看，非技术问题与技术问题并没有本质的区别，它们同样重要，也同样困难。

**提示：** 非技术问题也十分重要！千万不可掉以轻心。

## 11.1 答题要点

非技术问题虽然形式简单，但并不好回答，因为它们的答案没有对错之分。每个求职者的回答都不一样，每位面试考官想听的回答也不相同。帮助求职者回答非技术问题的书有很多，不少作者甚至为特定的问题准备了特定的答案，连什么时候应该朝面试考官点头微笑都替求职者考虑到了。

我们不想在这里重复那些老生常谈，本章内容将主要集中在技术类面试中最为常见的非技术问题上。

非技术问题的目的是了解求职者的个人经验和工作能力，并与其他的求职者进行对比。所谓个人经验既包括了你的工作经历，也包括了你的知识面和知识水平。有时候，即使你在回答技术类问题的时候表现优异，如果你以前的个人经验与有关职位不相称，你也有可能得不到这份工作。因此，在回答有关个人经验方面的问题时一定要慎之又慎——这类问题通常是面试考官怀疑你能否胜任有关工作的一个信号。此时，扬长避短、不卑不亢将是你最应该遵循的原则。举个例子：假设面试考官问你："你使用过Solaris吗？"——你的面试考官已经看过你的简历，因而应该知道你没有这方面的经验。那么，他的这句话可以这样来理解："我们用的是Solaris。你能在没用过Solaris的情况下胜任这份工作吗？"所以，不要仅回答"没用过"。你应该先强调一下类似的经历。比如说，你可以这样回答他："我没用过Solaris，但我在很多种操作系统上使用过很多种开发工具，所以我对多接触一种新的操作系统并不感到困难。我是一个不依赖于操作系统的开发人员。"当有人向你介绍工作职责的时候，一定要注意听讲。在做自我介绍的时候，要突出类似或者相关的工作经验，让自己比别人更有竞争力。

在回答非技术问题的时候，你还应该努力表现出你的"随和"来。"随和"与你能否融入该公司、能否成为一名有贡献的成员有很大的关系。有不少人认为"随和"就是"好脾气"，但这只对了一半——能否与同事友好相处、相互协作也是重要的一环。举个例子：你可能会说出这样的话："在前一家公司，我设计并实现了一个系统，把我们公司的人事资料都搬到Web上去了；根本没用别人帮忙。"这句话听起来像是在夸耀自己，但对方却可能会因此而怀疑你是否善于与同事合作。因此，要注意展示你的团队精神。你应该表达出这样一种愿望：希望加入一个优秀的团队并成为一名有贡献的成员。"团队"是一个人人爱听的词——没有人不爱听。

**提示：** 非技术问题的重要目的是为了判断你有没有相应的经验以及能不能融入他们现有的团队。

在阅读以下的示例问题及有关讨论时，请大家也为自己准备一份"标准"答案。请考虑你自己应该如何回答这些问题；在不同的场合，应该突出自己的哪些方面。现在做好准备，等到了面试考官那儿就胸有成竹了。如果在面试现场发现自己准备的"标准"答案不够有利，千万不要慌张，一定要懂得随机应变。最后一句忠告：要让自己的每一个回答都表现出你将是一位有价值的员工。

## 11.2 问题：你打算从事哪方面的工作？

一定要注意是谁在问你这句话。如果问话的人是位人力资源代表，请老老实实地告诉他你想从事什么工作。那位代表通常会根据你的回答而安排你参加相应的面试。

如果问话的人是位技术型的面试考官，那可千万要留神——如果答得不好，你可能就得不到这份工作了！这类面试考官提这个问题的目的通常是为了了解你的目标和志向。如果你对这个问题的回答与对方提供的职位有差异，对方就可能会建议你去另找一份工作。因此，如果你想获得这份工作，就必须表现出你的兴趣来；态度要诚恳，最好还能给出一个理由来。比如说，你可以这样回答："我一直对系统级的程序设计感兴趣，也愿意从事这方面的工作。所以我想加入一家大公司去搞一些系统级的工作。"或者，你可以这样说："我想从事Web程序设计工作，这样我就可以让朋友们看到我的工作成果了。我希望能在一家新成立的公司里做这些事情，我想这能更好地发挥我在Web服务器方面的经验并帮助公司更快地成长。"

有时候，你对自己想要从事哪方面的工作并不太清楚。此时，你应该把该公司描绘成你最向往的地方。你可以泛泛地说你想从事开发工作，它既令人激动，又有很多的表现和学习机会。你还可以说你认为具体工作只是一个方面，更重要的是他们优秀的团队和前途无量的公司。这类回答一般不会有太大的毛病，不至于把这份工作给"说"跑了。

注意，有理智的热情与盲目的渴望是有区别的。谁都不愿意聘用一位被其他公司屡屡拒之门外的人。千万不要像这样回答："你让我干什么工作，我就干什么工作。"这类回答的结局往往就是一封"谢谢你来参加面试"之类的信函。

另一种可能是你确切地知道自己想从事哪方面的工作，并且不愿意接受其他类型的职位。如果真是这样，就不要在你不愿意接受的工作方面多说话。这可能会让你得不到某些工作邀约，但反正是你不喜欢的工作，得不到也没什么。明确地表明自己的想法有这样一个好处：假如你最先走进去的面试办公室没有你感兴趣的职位，那你最后走进去的面试办公室就可能有你感兴趣的职位了。

关于这个问题，我们还有最后一句忠告：这是你明确地表达自己希望加入一个优秀团队的好机会——不要避而不谈。能否加入一个优秀团队的机会就在你自己手里。

## 11.3 问题：你最喜欢的程序设计语言是哪一种？

这似乎是一个技术型的问题，里面也确实有些技术含量——你需要给出一些具体的技术性理由来解释你为什么喜欢某种语言。但这个问题也包含着一些非技术性的东西。有不少人对自己喜欢的程序设计语言、计算机、或者操作系统有一种宗教情结。这些人往往很难共事，因为他们总是坚持使用他们最喜欢的东西，哪怕那些东西并不适合用来解决手里的问题。你应该避免让别人认为你是一个这样的人。要知道，你最喜欢的程序设计语言也有它的局限性，不适用于某些问题。在回答这个问题的时候，你应该先把自己熟悉的程序设计语言都列举出来，然后表明这样一个观

点：不同的程序设计语言适用于不同的软件项目，根据项目需要去选择最合适的程序设计语言才是最重要的。

这一原则适用于其他类似的“最喜欢”问题，比如：“你最喜欢的计算机是哪一种？”“你最喜欢的操作系统是哪一种？”等等。

## 11.4 问题：你的工作习惯是怎样的？

这个问题往往是这样一个信号：你应聘的这家公司有一些不同寻常的工作习惯。比如说，这家公司刚刚起步，需要在比较差的环境里长时间工作，或者，这是一家大公司，但刚启动一个新项目。简而言之，如果想回答好这个问题，一方面要在事先把自己的工作习惯弄清楚，另一方面要把对方的工作习惯弄清楚。

## 11.5 问题：可以说说你的个人经历吗？

每位求职者都应该为这个问题准备一个“标准”答案。请大家务必重视这个问题！在回答这个问题的时候，你一定要把自己以前的成绩突出出来，在谈到自己参加过的项目时要充满热情。不仅要有实实在在的业绩，也要有总结出来经验教训。干成功了的事情要说（你是一位成功者），干失败了的事情也不妨说几件（谁都不可能不犯错）；不要忘记把成功的经验和失败的教训总结出来。根据你个人的情况，请把回答这个问题的时间掌握在30到60秒。另外，一定要在事先做些练习。

## 11.6 问题：你的职业目标是什么？

你可以利用回答这个问题的机会来解释一下你为什么想得到这份工作（金钱以外的理由）以及这份工作与你的目标有什么吻合。这与“你打算从事哪方面的工作？”的问题有些类似，但这两个问题有着细微的差异：如果这份工作与你的职业目标有冲突，对方就可能会怀疑你不想做这份工作。如果你说不上来自己的职业目标，没关系——很多人都说不上来。不过，既然被问到了，你至少应该说出点方向吧。你可以简单地这样说：“我希望能先参加一些大项目的开发工作，然后往项目管理的方面努力。再往后，就很难说了。”这个回答一方面给出了一个目标（虽然不够远大），另一方面也能让对方增加点你能胜任这项工作的信心。

## 11.7 问题：你为什么要换工作？

面试考官通常都希望知道你不想做或者不喜欢哪些事情。你显然是不喜欢前一份工作，要不你就不会来参加这个面试了。当然，还有一种可能是你因为某些难言之隐而不得不放弃前一份工作。不管怎样，在回答这个问题的时候，你应该把理由归结到环境的变化，不可抗力，或者面试考官已经知道的某个弱点上。比如说，环

境变化方面的理由可以这样说："我在一家大公司里工作了五年，眼看着软件行业的发展。但我不想再待在大公司里了，我想成为一家新公司里的关键人物，从起步开始帮助它发展起来。"或者，你可以这样回答："我以前在一家新公司里工作，但那儿的管理太乱了。所以我想换一家管理好的公司上班。"

不可抗力方面的理由可以这样说："我现在上班的公司放弃了我一直在搞的项目，又准备把我安排到一个我没多大兴趣的岗位去。"或者，你可以这样回答："我所在的公司被收购了。从那时起，整个气氛都和以前不一样了。"

用一个面试考官已经知道的弱点来回答这个问题也是可以接受的。比如说："我的前一份工作需要做大量的系统级程序设计。这一点不是我的强项，我的兴趣也不在于此。我对Web程序设计更感兴趣，而且在这方面也有很丰富的经验。"

最后一句忠告：虽然金钱也是更换工作的一个好理由，但最好不要把这一点当做主要理由来强调。对方很可能会产生这样的怀疑：既然你想挣更多的钱而你的前一位老板却不情愿给你加薪，是不是因为你没有这样的价值呢？你是不是在试图掩饰这个事实呢？

## 11.8 问题：你希望拿多少报酬？

这个问题可能出现在任何场合，但最为常见的场合有两种：在面试的刚一开始或者在该公司决定给你这份工作的时候。如果这个问题出现在面试的开始阶段，那么，要么对方是想根据你对收入的预期来判断有没有必要与你做进一步的谈话，要么是对方对有关职位的工资行情不太了解。一般说来，你应该尽量把这个问题往后推——在你向对方证明你的价值之前，为工资高低而讨价还价是没有意义的。如果你在面试初期回避不了这个问题，那最好是给对方一个你对收入的预期范围，把你心中的数字做为这个范围的底限，这对今后的讨价还价会有好处。

如果对方是在面试接近尾声的时候才向你提出这个问题的，那就只能是一个好信号。如果到了这个时候对方还没有决定聘用你，他们就不会向你提出这个问题。一般说来，大公司对这类事情不像小公司那么斤斤计较。而这个问题通常也表明对方愿意就工资的具体数额与你进行商谈。请注意，很多公司并不清楚怎样才能开出一个有吸引力的价码来，你应该把握这个机会告诉对方像你这样的人才值多少钱。

首先，请务必在事先做一些调查研究。如果你发现同类职位的年收入是$40 000到$55 000美元，你的年收入达到$80 000美元的希望就不大。其次，千万不要把自己给"贱卖"了。如果你希望自己的年收入能达到$70 000美元，就不要告诉对方说你想找一份年收入$60 000的工作并期望对方会往上涨。第三，你应该对收入的构成有一个全盘的考虑。比如说，如果你刚从大学毕业，需要支付找公寓、搬家、买汽车的首期款，等等，那你可能希望有一笔签约金来支付这些费用。或者，如果你正准

备加入一加刚起步的公司，那你可能希望有较好的股票期权，工资低点倒没多大关系。不管怎样，你应该把奖金、福利、股票期权、工资等放到一起做全面的考虑。

一般说来，最好不要把你的底牌过早地暴露给对方。谁掌握的信息更多，谁在谈判中的位置就越有利。在直接回答有关工资收入的问题前，最好先问问面试考官他们打算支付多少钱。他的回答将不外乎以下四种可能。

第一种，对方给出的范围恰好与你期望的一样。此时，如果按照下面的原则进行，就通常能把这个数字抬高一点儿。首先，不要显得太激动——保持镇定。第二，告诉对方说你的想法比他们给出的范围略高一点，把你心目中的最低值做为对方的最高值。比如说，如果对方说："我们打算支付$40 000到$45 000美元"，那你应该这样回答："这听起来很不错。我正好想找一份年收入为$45 000到$50 000美元的工作，但我希望能达到这个范围的高点。"第三，在就具体数额达成一致之前，请保持良好的谈判风度。这些原则基本上能保证让你拿到$43 000到$48 000美元的年收入。

第二种，对方开出来的价码比你预期的要高。这就没什么好说的了。

第三种，对方没有回答你的问题。他可能会这样说："根据求职者的具体情况，我们有很宽的工资范围供他们选择。你是怎么想的呢？"这其实是一个好信号，表明他可能有权决定是否应该向你支付更高的工资。这个回答一方面表明对方愿意与你谈判，另一方面也表明对方是一位精通此道的谈判高手。此时，在做好心理准备之后，你应该把自己预期范围的最高值说出来——只有给自己留出谈判空间才能得到对你有利的邀约。比如说，如果你的预期范围是$55 000到$60 000美元，可以这样说："我希望是每年$60 000美元。"与给出一个范围相比，如果你说出的是一个准确的数字，就不会给对方留下太多讨价还价的余地。要避免下面这样的回答："我希望……"或者"我比较喜欢……"。对方可能会接受你的数字，也可能会提出一个略低的数字来。如果你保持着良好的风度并耐心地进行谈判，最终的数字肯定会落在你预期的范围内。不过，说不定对方会反过来提出一个比较低的工资范围。此时，你应该按下面介绍的第四种情况做出反应。

第四种情况是对方提出的数字比你的期望值低。此时，你可以利用下面几种技巧来设法提高你的工资数额。首先，再次强调你的工作能力并重申你的收入预期范围。比如说，如果你的预期值是$50 000美元而对方提出来的数字只有$35 000美元，那你就可以这样说："我得承认，这个数字比我想像的要低得多。根据我在Web开发方面的经验和我将对贵公司做出的贡献，我期望能够获得$50 000美元的年收入。"如果对方说还需要再考虑考虑才能答复你，这是很可以理解的。如果对方在听到你说出来的数字后仍不打算提高你的工资，他们通常会提出以下几种理由：

1) 你给出的数字超出了他们工资预算。

2) 公司其他员工的工资都没有这么高。

3) 你的个人条件不能说服他们给你这么高的工资。

这些理由都是站不住脚的。首先，工资预算是相对于整个公司而言的，与你个人的工资高低没有关系。如果这家公司真的需要你的加盟，它们就肯定能拿出这笔钱，也肯定能克服工资预算之类的人为障碍。如果它确实是拿不出这笔钱，那你还是别加入一个如此捉襟见肘、日暮西山的公司好了。

其次，公司其他员工的工资与你并没有多大关系，那是他们与公司之间的事。不应该拿其他员工与你来做对比。你可以这样来回答对方："我想，我的收入与其他员工的收入是两回事，把它们牵扯到一起好像不合适吧。我希望能得到一份与我个人能力相称的收入，并且相信$XXXX就是一份这样的收入。"

最后，如果你在事先做过调查研究的话，就应该知道自己的经验和技巧配得上你说出来的工资数额——对方只是想再把它压得低一些而已。你应该再次强调你的经验和能力，并告诉对方：根据你所掌握的情况，你并没有漫天要价。对方要是知道市场行情的话，就应该会遵循市场规律并提高它报出来的数字。

如果对方没有增加你的工资而你却仍然希望得到那份工作，你还有两个最后的撒手锏。第一，你可以说自己很中意那份工作，但希望有六个月的试用期，等到时候再根据你的表现来最终确定你的正式工资。不过，一般说来，在没有参加那个公司之前，你说的话往往更有效力。所以你也别对六个月后的情况抱太多的幻想，但大多数面试考官都会同意你的这一要求——千万要记得把这一点写成文字。其次，再就其他方面做些争取。比如说，你可以在带薪假期、灵活的上班时间、或者签约金等方面再做些努力。

下面是关于工资收入问题的最后几句忠告。首先，有些害羞的人不习惯为工资的事讨价还价。但你要清楚，找工作其实是一种商业行为，工资也是这一行为中的一个环节。谁都不会白干工作不拿钱，这一点对方也知道；所以你没有必要把你自己表现得好像不在乎工资收入似的。

有些面试考官会用福利、工作条件等东西来吸引你加入他们公司。这些因素当然是找工作时必须考虑的东西，但你仍应该把全部的报酬条件都谈清楚才好。记住，像福利、工作条件之类的东西通常都不会有很大的谈判余地，所以你也不必为了这些无法探讨的东西而浪费谈判精力——千万不要让对方用这些次要因素分散了你的注意力。

## 11.9 问题：你以前的报酬水平是多少？

这个问题其实是"你希望拿多少报酬？"的另一个问法。对方是想通过了解你以前的收入情况来确定应该给你多少报酬。在遇到这个问题的时候，（除非你对自己以前的收入很满意）应该这样来表达：你的报酬应该与新工作的职责相称，与你担

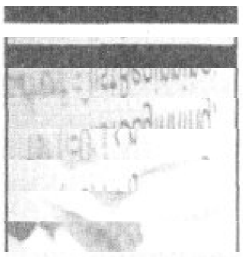
任其他工作时的收入水平没有必然的联系。另外，千万不要夸大自己以前的工资收入，要不然，如果对方让你拿出证据来，你可就无法收场了。

## 11.10 问题：我们为什么要雇佣你?

这个问题很粗鲁，多少还有点挑衅的味道。这个问题的潜台词是：没有足够的理由能表明你将胜任那项工作。很明显，你的经验和能力应该是符合要求的，要不然，面试考官就不会和你进行这场谈话了。此时，千万不要单纯地采取防守姿态并拿出自己的简历和对方进行争论。你应该以一种正面的态度来对待这个问题，比如说，谈谈你为什么想加入那家公司，那项工作对于你是多么的适合。这样的回答既表明了你善于应付不利局面，又能不露痕迹地“锉”一下那位无礼的面试考官。

## 11.11 问题：你有什么问题想问我吗?

常识告诉我们，你应该在此时提出一些问题以表明自己对新工作的热情和对面试考官的尊重。但如果面试进行得很成功，在最后关头问一个愚蠢的问题就将是一个不可饶恕的举动。因为是你而不是对方在参加面试，所以一定要挑一个不会对你产生不利影响的小问题来问。一个深思熟虑的好问题既能让你了解到该公司更多的情况，也能给面试考官留下一个更良好的印象。一般说来，你的面试考官不会主动告诉他所担任的职务，所以你不妨利用这个机会来问一下。这个问题既能让你对自己今后的工作情况多一点认识，也表现出了你对面试考官本人的尊重。此外，如果面试考官在面试过程中提到过一些有趣的事情，你也可以趁此机会请他再详细地说一说，这有助于你进一步了解你未来的上司。最后，如果你实在想不出应该问些什么，不妨开个无伤大雅的小玩笑——你可以这样说：“我本来是有好多问题的，可今天早上的那位面试考官已经都把它们回答完了。有什么事等我们成为同事以后再问你吧！”



# 附 录

## 写个人简历的方法

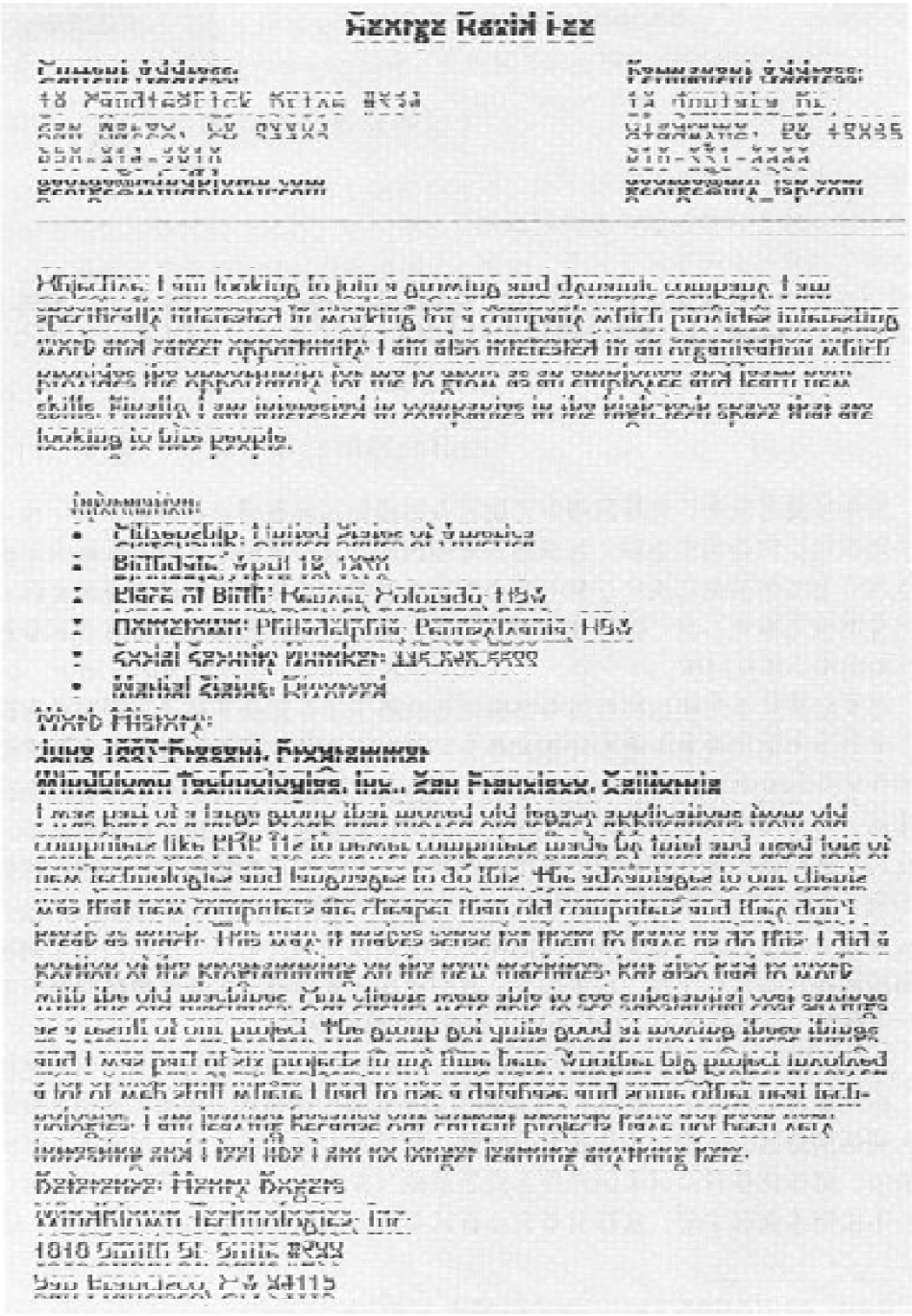
无论是通过熟人、参加招聘会、猎头公司还是其他渠道，只要是想找工作，你就必须把自己的简历准备好。你的简历必须向别人证明你的工作经验和能力足以让你成为一名合格的候选人。如果对方没有在你的简历上发现他们感兴趣的东西，就会把你的简历撇在一边，把你遗忘到脑后去。总之，个人简历非常重要，它是你找工作时的第一块敲门砖。

用来应聘技术类职位的简历与各种简历教科书里介绍的非技术类简历是有区别的。非技术类简历通常不需要详细列举各种技能，而技术类简历则需要求职者把自己的技能详细地列举出来。如果求职者不具备必要的技能，对方就不会有兴趣找他来谈话。这意味着技术类简历要比非技术类简历更详细，更具体。在本附录里，我们将以一些应聘软件开发职位的简历为例对个人简历的书写技巧进行介绍。我们挑选的第一个例子是一份写得非常糟糕的个人简历（如图A-1所示），它的主人是一位初级开发员。虽然我们希望读者写的简历不会糟糕到如此地步，但我们用来润色这份简历的技巧却适用于每一位求职者。我们之所以要挑选一份如此糟糕的简历做为第一个例子，目的就是为了能够把更多的技巧介绍给大家。

这份简历中的大部分问题都犯的是同一个低级错误。Lee的这份简历写的是他自己，而不是为了找工作。说白了，Lee的简历更像是一份自传，根本不能用来推销他本人和他的经验。这是一个很普遍的问题，有很多人都有这样的心理误区：既然叫做简历，就应该把自己过去的经历全都简单地写在上面；这样，招聘方就能对自己有一个比较全面的了解，就能对是否让自己参加面试做出正确的决定。可惜，实际

---

译者注：我们没有对本附录中的示范简历进行翻译。因为1）附录中的讨论必须用它们做为示例；2）我们希望大家对软件开发类职位的英文简历也有所了解，这类职位的申请人简历往往要求是英文的。



Objective: I am looking to join a growing and dynamic company. I am specifically interested in working for a company which provides interesting work and career opportunity. I am also interested in an organization which provides the opportunity for me to grow as an employee and learn new skills. Finally, I am interested in companies in the high-tech space that are looking to hire people.

- Citizenship: United States of America
- Birthdate: April 12, 1977
- Place of Birth: Kansas, Colorado, USA
- Hometown: Philadelphia, Pennsylvania, USA
- Social Security Number: 123-45-6789
- Marital Status: Married

Work History:

**June 1997–Present, Programmer**

**Middlemiss Technologies, Inc., San Francisco, California**

I was part of a large group that moved old legacy applications from old computers like PDP-11s to newer computers made by Intel and used Unix as their operating system. The advantages to our clients was that new computers are cheaper than old computers and they don't break as much. This way it makes sense for them to have us do this. I did a variety of jobs, including porting all the new machines, but also had to work with the old machines. Our clients were able to see substantial cost savings as a result of our project. The group got quite good at moving these things and I was part of six projects in my time here. Another big project involved a lot of web stuff where I had to use a database and some other new technologies. I am leaving because all current projects have now been completed and I feel like I am no longer learning anything here.

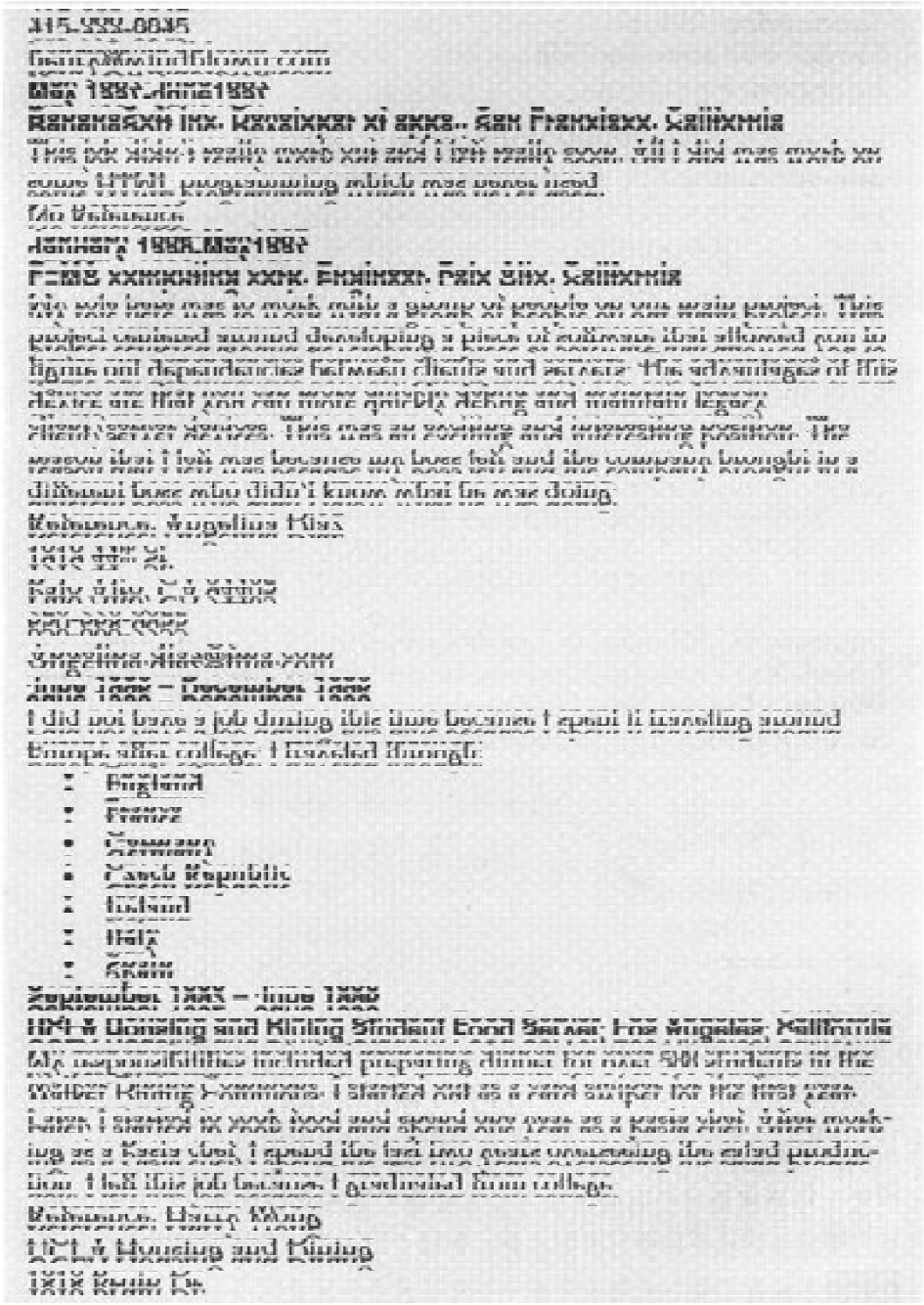
References: Henry Rogers

Middlemiss Technologies, Inc.

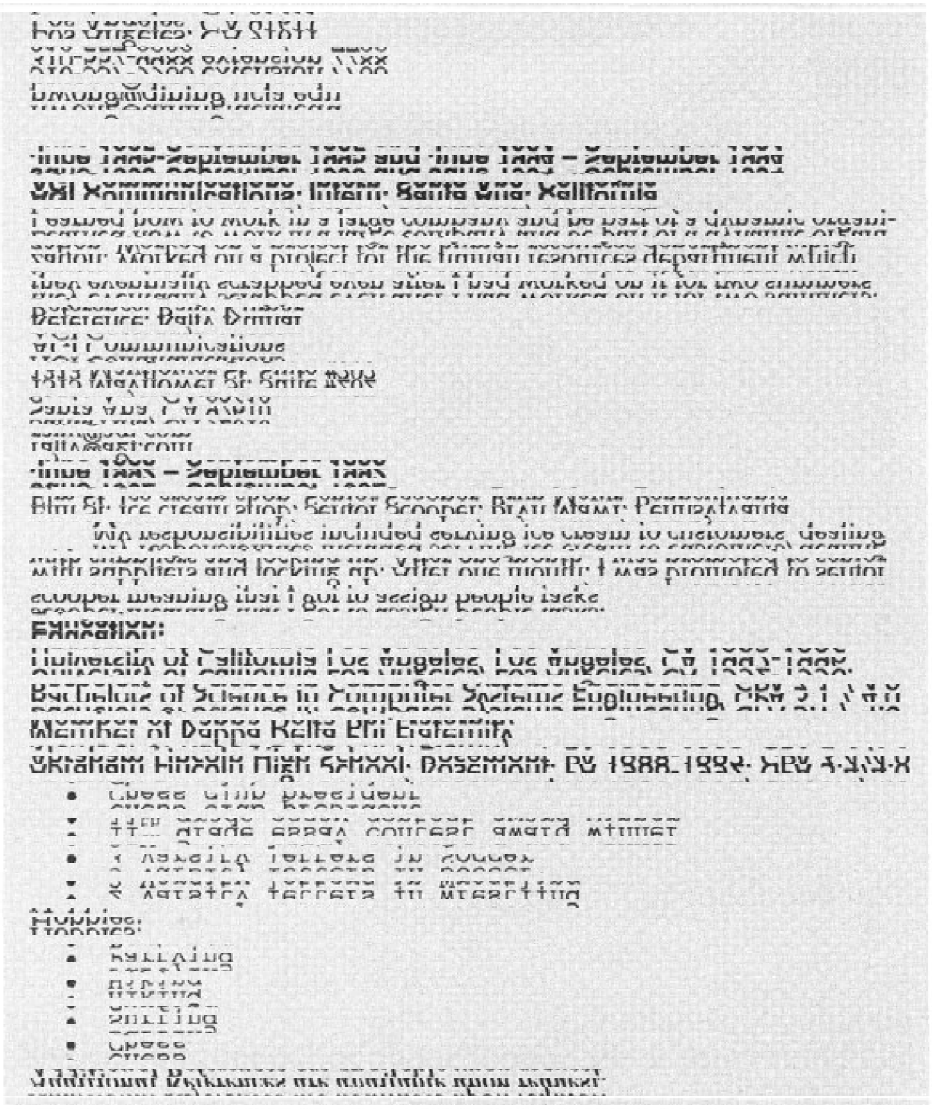
1818 Smith St. Suite #233

San Francisco, CA 94115

图A-1 润色前的个人简历（初级开发员）



图A-1 （续）



图A-1 （续）

情况却根本不是这样——招聘方并不会花太多的时间去详细阅读他们收到的每一份简历。因此，你必须把自己的简历作为一个能够把你本人尽快推销出去的工具，一个能够让招聘方尽快认清你的价值的工具。明白了这一点，其他问题就不言自明了。

**建议：** 要把自己的个人简历作为一个有效的推销工具。

Lee的简历存在着一些很普遍的问题。首先，篇幅过长。面试考官收到的简历往

往会超过50份。根据以前的经验，他知道能够符合有关条件的应聘人不会有很多。面试考官的时间通常只够用来与四到五位候选人进行谈话，所以他必须把他手里百分之九十的简历排除在考虑之外。面试考官并不会花太多的时间去仔细阅读每一份简历，他们会迅速浏览每一份简历，看里面有没有给出值得他把这份简历留下来做进一步考虑的理由。此时，面试考官关心的是这样一个问题："这个人能为我们公司干些什么？"你必须把自己的简历写得让面试考官不"忍心"把它丢到一边去才算合格。要知道，面试考官在剔除求职者的个人简历时并不需要做痛苦的内心斗争。如果他没有在求职者个人简历的第一页里看到让他感兴趣的东西——注意，这段时间往往只有15到20秒——就会毫不迟疑地拿出下一份简历。

你当然希望自己的简历能够给人留下深刻的印象，但千万不要为此而自我吹嘘或弄虚作假。这会给你自己带来很多麻烦：首先，很多面试考官都会根据你的简历来提问题，如果你回答不上来或回答得不够好，就会让他对你整个简历产生怀疑。其次，如果你吹嘘得过了头，面试考官根本不需要对你进行面试就能判断出你是在撒谎。最后，你的不实之词会让你显得像个"万金油"，什么都知道一点，但却什么也拿不出手。结局是可想而知的：你的简历不但没有把你推销出去，反而成为你得到一份好工作的障碍了。

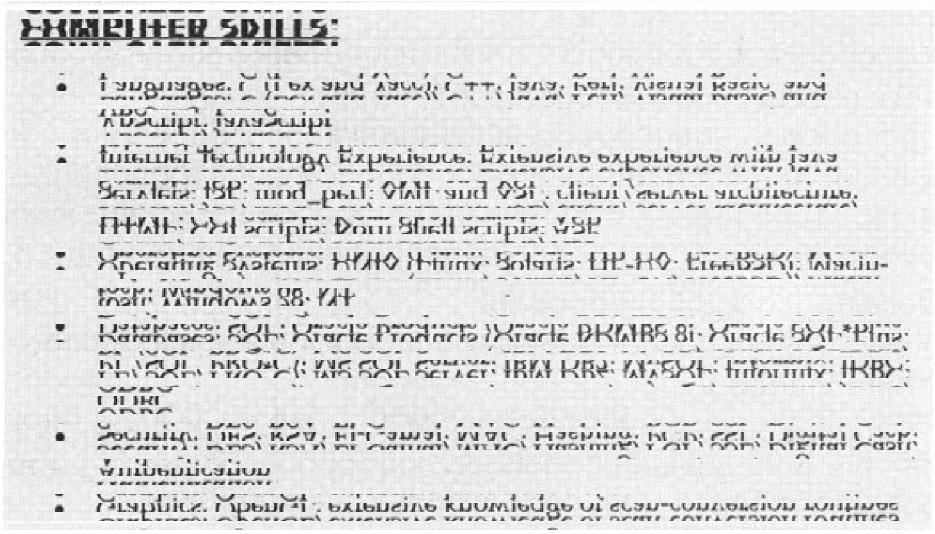
写简历的基本原则之一是尽可能地短小精悍。一般来说，如果你的工作经验少于5年，你的简历就应该保持在一页纸以内；如果你的工作经验少于15年，你的简历就应该保持在两页纸以内。无论如何，你的简历都不应该超过三页纸。


个人简历应该写得短小精悍，要让每个字都发挥作用。

压缩个人简历篇幅的办法并非只有删节信息一种。就拿Lee的简历来说吧，它里面是有不少东西应该被删节掉的，但大部分内容只是需要改进而已，它甚至还需要我们再增加一些东西。

从内容上讲，Lee的简历里里并没有不实之词，但它缺少"技术"——没有使用科技名词。这对他可是一个大问题。有些公司会用扫描仪把收到的个人简历扫到一个计算机系统里，并给它们加上关键词索引。这样，当公司需要招聘一位"有XML经验Java开发员"时，系统就会把带有单词"Java"和"XML"的个人简历给找出来。其他一些公司会按技能对个人简历进行分类，但最终效果却都差不多。因为Lee的个人简历里缺少科技名词，所以它很可能都递不到面试考官的手里去。首先，Lee应该把自己使用过的各种软件产品、操作系统、程序设计语言等都列举出来，再把自己曾经参加过的有关项目——比如安防算法或网络协议等——也列举出来；然后再把这些东西有条理地组织到一起。Lee应该按有关领域对自己的技能进行分类，如图A-2所示。

当你在自己的个人简历里列举产品名称的时候，应该把各种产品的具体版本也列出来，这表明你能跟上时代的步伐。在润色Lee的简历时，我们没有列出相应的版本号，因为等大家读到这本书的时候，这些版本号可能早就过时了；但我们希望读者在写自己的个人简历时不要忘记这一点，个人简历应该比书更新速度快才对。你应该把自己的进步随时更新到自己的个人简历里去。与其在个人简历里列出一个过时的版本号，还不如光写出一个产品名称而不写出它的版本呢。



图A-2 初级开发员示范简历的“个人技能”部分

**提示：** 在个人简历里，一定要列出技能名称。

Lee的个人简历还需要在排版方面下些功夫。首先，Lee的简历现在使用的字体、格式和段落实在是太多了，很多人都不喜欢看这类东西——它更像是一份草稿。其次，自动扫描系统可能无法处理这种简历。Lee应该只选择一种字体（比如Times字体），并在整个简历里只使用这一种字体。

Lee的简历很不容易阅读，条理不清晰，重点不突出，还没有把自己的贡献写出来，让人很难认为他将成为一名有价值的员工。这一点在他的工作经历部分尤其明显。首先，Lee应该用项目符号或编号把内容逐条地列出来，这一方面将比把它们写成一个大段落要更容易阅读，另一方面也有助于面试考官迅速了解Lee的情况。这将大大增加Lee的个人简历被留在面试考官手中而不是被撇在一边的机会。

Lee的简历还应该在突出重点方面下些功夫。Lee自己写的简历没有把他自己做过的事情写清楚。他写了团队的任务、公司的业务，却惟独没有写他自己在其中发

挥的作用，而这最后一点恰恰是他本人找工作时最重要的因素。首先，Lee应该用诸如“impletement”（实现）、“design”（设计）、“program”（编程）、“monitor”（监控）、“administer”（管理维护）、“artechitect”（规划）之类的动词来描述自己的贡献。这些动词能够准确地把他的工作性质反映出来，比如“designed database schema for Oracle8i database and programmed database connectivity using Java threads and JDBC”（为Oracle8i数据库设计了数据表结构，用Java线程和JDBC编写了数据库的应用接口），等等。其次，只要有可能，就应该用数字来量化自己的任务和工作成就。比如“administered network of 20 Linux machines for Fortune 100 client, resulting in $1 million in revenues annually”（为某《幸福》100强企业管理由20台Linux机器构成的网络，由此而产生的年净利润为100万美元），等等。这样写的推销效果就好多了，因为它很好地回答了“这个人能为我们公司干些什么？”的问题。需要注意的是，只把你认为能给人以深刻印象的数字写出来就行了，如果你的数字不那么引人注目，就别写在简历上。

如何安排有关职责的先后顺序也是需要动脑筋的。这里的一般原则是，先写最“唬人”的，后写最一般的；但一定要把与你现在申请的这份工作最有关的写在最前面。举个例子：假如你以前同时从事过销售和开发工作，你就应该有一些值得自豪的销售业绩、一些值得自豪的开发业绩、一些不起眼的销售业绩以及一些不起眼的开发业绩。那么，如果你现在应聘的是销售工作并希望突出自己的销售能力，就应该把那些销售业绩写在前面，把那些开发业绩写在后面。最后，一定要有条理，把同一主题的内容放在一起——不要教条地严格按时间顺序或者重要程度来排列。

很多人都不善于利用个人简历来推销自己。他们认为做人应该谦虚，不应该自吹自擂。但这类人的结局往往是把自己给“贱卖”了。人是不应该撒谎，但把自己实实在在的成绩用能给人以深刻印象的词说出来却没有什么不好。如果你真的不善于“吹捧”自己，就应该找一位了解你的朋友来帮忙。

**提示：** 一定要利用项目符号或编号把有关经历逐条地列出来，要尽量突出其中的正面因素。

Lee的个人简历里还有一些白白浪费宝贵篇幅的无关细节。比如说，面试考官最先看到信息之一是：Lee是一位美国公民，出生在丹佛市。就算与那份工作有关，Lee也没必要把自己的国籍和出生地放在最前面，面试考官最关心的首先是Lee能不能胜任那份工作。把这些东西放在前面等于白浪费考官的精力。其他诸如此类的东西还有出生日期、老家、社会安全号码、婚姻状况、业余爱好、个人旅游史，等等。这些东西并不能加重Lee的砝码。此外，Lee还在简历里使用了大量的“我”（I）；这很没有必要，因为他的简历说的当然应该是他本人的事情。Lee也没有必要列出证明人来。在考虑做出工作邀约之前，面试考官用不着与Lee提供的证明人打什么交道，

所以在个人简历里列出这些人名来是没有意义的。事实上，甚至连“References are avaiable upon request”（可提供证明人）这句话都不必写在个人简历上，因为这根本用不着特意说明。类似地，个人简历也不是解释自己为什么会离开以前那份工作的地方。这种问题应该留到面试阶段由面试考官主动来问——Lee是应该为此准备一份“标准”答案，但不应该把它写到个人简历里。Lee姓名中的“George”也应该省略，除非他平时总是被人称为“George David”。最后，对于那些有可能会对自己找这份工作产生不利影响的枝梢末节，就不要让它们出现在你为这份工作而准备的个人简历里了。比如说，不要在个人简历里写上“在六月份毕业之前，希望从事兼职工作；此后，可考虑全职”之类的话——大多数面试考官都会把这类简历放到一边，转而去挑选一位能够出满全勤的候选人。不过，如果你能在面试阶段给面试考官留下了深刻的印象，事情就有商量余地了。

个人简历应该只包含必不可少的信息，要让它尽可能地短小精悍，要让每个单词都有作用。拿Lee的简历来说，他应该只给出一个联系地址——“现住址”、“永久地址”之类的东西只会让面试考官无所适从；这里的原则是：一个地址、一个电话号码、一个E-mail地址。此外，Lee把自己在中学时的情况写得太多了。这里的原则是：不要把与找工作无关的旧奖状、旧成绩单、旧职务之类的东西写在个人简历里；10年以前或者与现在找的这份工作无关的工作经历只要一笔带过就足够了。比如说，Lee花了不少笔墨来写他在冰激凌店和食堂的工作经历——这些事情不是不能说，但在冰激凌店的工作表现再好，Lee也不可能因此而得到眼前的这份工作。Lee只需把与眼前这份工作有关的经历写出来就足够了。此外，Lee不应该写出他只干了两个月的那份工作，这对他没有什么好处。最后，Lee的求职陈述写的也有问题——谁不想在一家“有活力的”公司找一份“有趣的”工作？Lee应该在求职陈述里简单地说明一下自己想找什么样的工作，比如“软件工程师”或“数据库程序员”等。

**建议：** 个人简历应该只包含与正在找的工作有关的信息。

在对内容进行了筛选润色之后，Lee还需要对这些信息的先后顺序做好安排。比较常见的办法是按时间顺序来排列。也就是说，Lee的简历要按高中、冰激凌店、大学……的顺序来写。这样，招聘方就很容易了解Lee的成长和工作经历。不过，虽然这是一种传统套路，却不适用于Lee的情况。求职者应该把自己最具竞争力的优势写在个人简历的最开始。因为面试考官将从头开始阅读求职者的个人简历，所以求职者就应该把自己最好的东西放在简历的最前面，用它们来吸引面试考官去阅读简历的其他内容。然后，再按一种清晰简明的顺序把自己的各种优势写出来；越往后，内容的重要性就越小，这样一直写到简历的末尾。因为求职者最近的工作经历与他正在找的工作往往最有关系，所以如果你想按时间顺序来进行排列的话，就应该按时间的倒序来写。

回到Lee的简历上。Lee的最大优势无疑是他的技能，而他的相关技能还真不少。Lee应该把这些技能写在简历的最开始，然后是他的工作经历或求学经历。如果刚从学校毕业不久，求职者就应该把自己的求学经历放在最前面——尤其是你毕业于著名院校时。如果你已经有了一定的工作经历，就应该把这些内容放在最前面。就Lee的情况而言，把哪部分内容放在前面还真不好决定。他正好处于把哪个部分放在前面都可以（或者说都不可以）的情况。Lee刚从一所著名学校毕业不久，也有了一定的工作经验，只是上班的历史还不很长而已。这么考虑的话，把求学经历放在工作经历的前面应该会比较好一些。Lee的求学情况很简单，只有一个条目。如果他有两个或两个以上学位的话，就应该把最“唬人”的那个放在最前面。

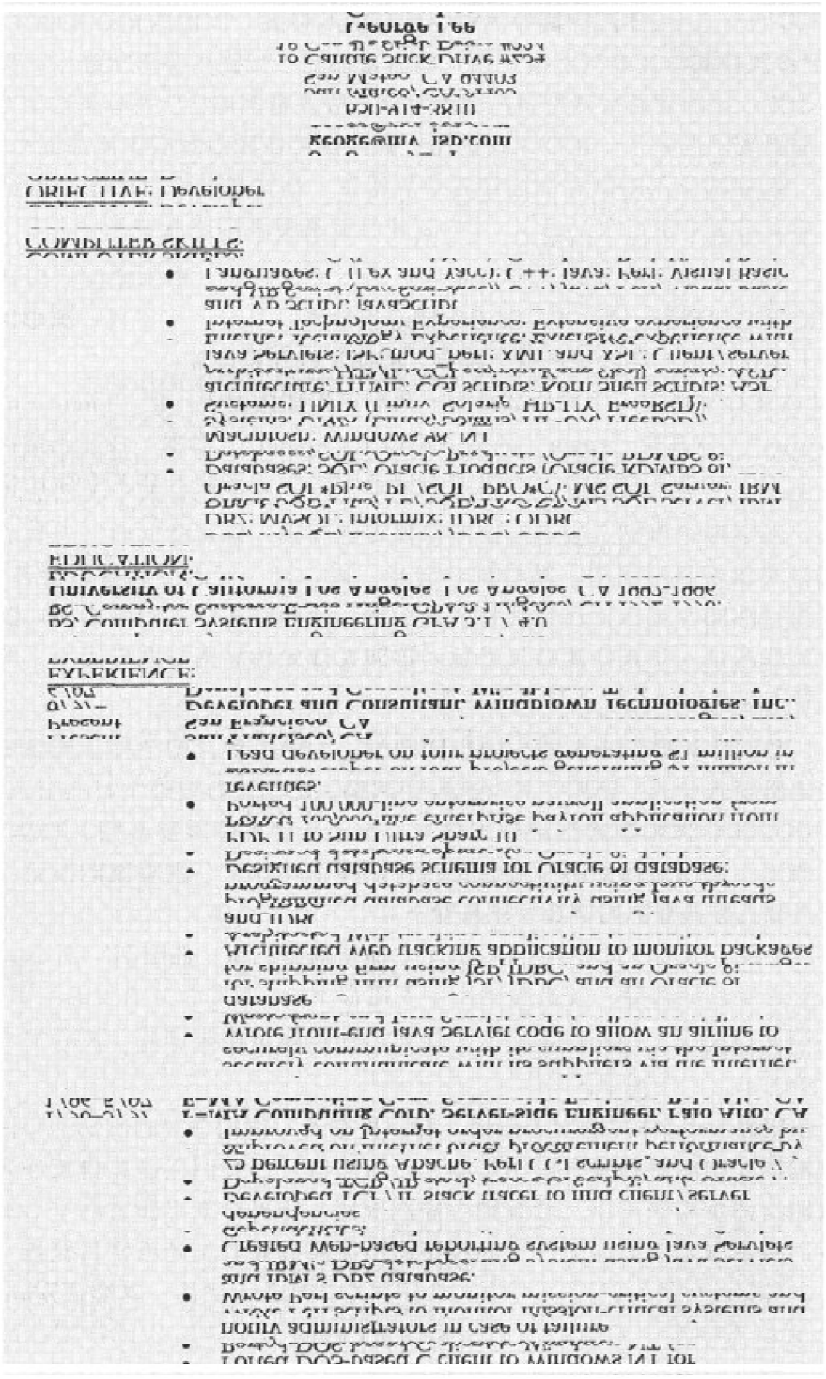
Lee还应该仔细检查一下简历的拼写。我们发现的错误有：把“interesting”错拼为“intresting”、应该用“spent”的地方错用了“spend”，等等。这些小错误会给人一种粗心大意、不堪大用的感觉。很多面试考官只要在求职者的个人简历里看到有拼写错误，就会把它丢到一边去；即使没被丢到一边，也会降低面试考官对他的印象分。请反复检查自己简历；然后把它放在一边，过一段时间后再来检查它几次。请一位信得过的朋友来帮忙检查也是个好主意。在朋友读完你的简历后，问他还有什么地方需要改进，是否还能做得更好。你朋友读完你简历后的反应多少能让你对面试考官看完你简历后的反应摸着点线索。

最后，我们还想说说个人简历的打印问题。一般说来，如果你是通过电子邮件把简历发送给招聘方的，就用不着考虑打印问题。如果需要由你来打印自己的简历，用不着使用特殊的纸张或者精密打印技术。个人简历可能会被复印、扫描、传真、写上字，等等，使用特殊的纸张或者精密打印技术无异于一种奢侈的浪费。有一台激光打印机和一些普通的白纸就已经足够了。

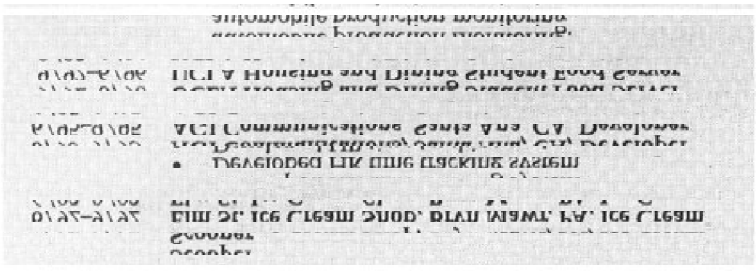
在经过了这么多的改进之后，Lee的个人简历焕然一新，如图AP-3所示。

正如大家看到的那样，这份新简历大大增加了Lee走进面试考官办公室的机会。新简历与旧简历的主人是同一个人，工作经历和技能也还是那些，但它们之间的差别却是一个天上，一个地下。

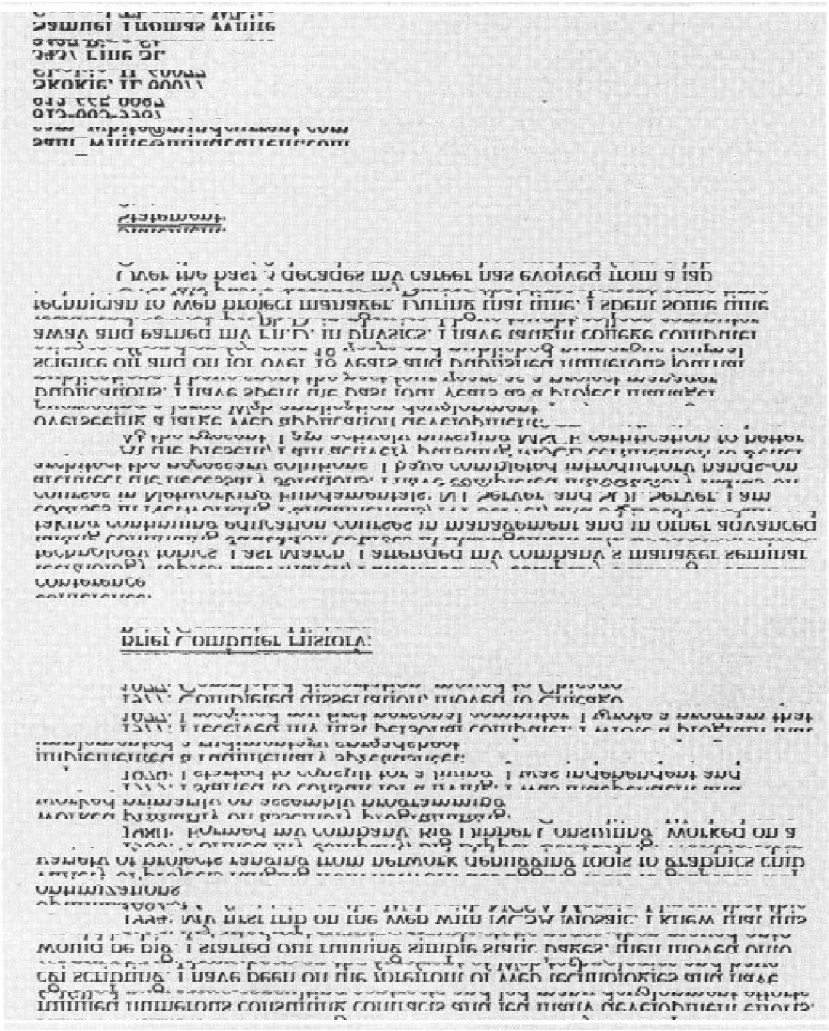
润色完初级开发员Lee的个人简历，我们再来看一份用来应聘高级职位的个人简历。虽然前面介绍的思路也可以用来润色这份简历，但我们还需要再多注意几个方面。高级职位通常需要承担一些管理方面的责任，所以这类求职者的个人简历还必须证明他们具备有相应的能力。下面这份简历应聘的职务是高级项目经理，它的主人名叫Sam White。图A-4是Sam White未加润色前的个人简历。在阅读White的简历时，请大家留意一下，看我们前面用来润色Lee的个人简历的技巧哪些能够帮到White。



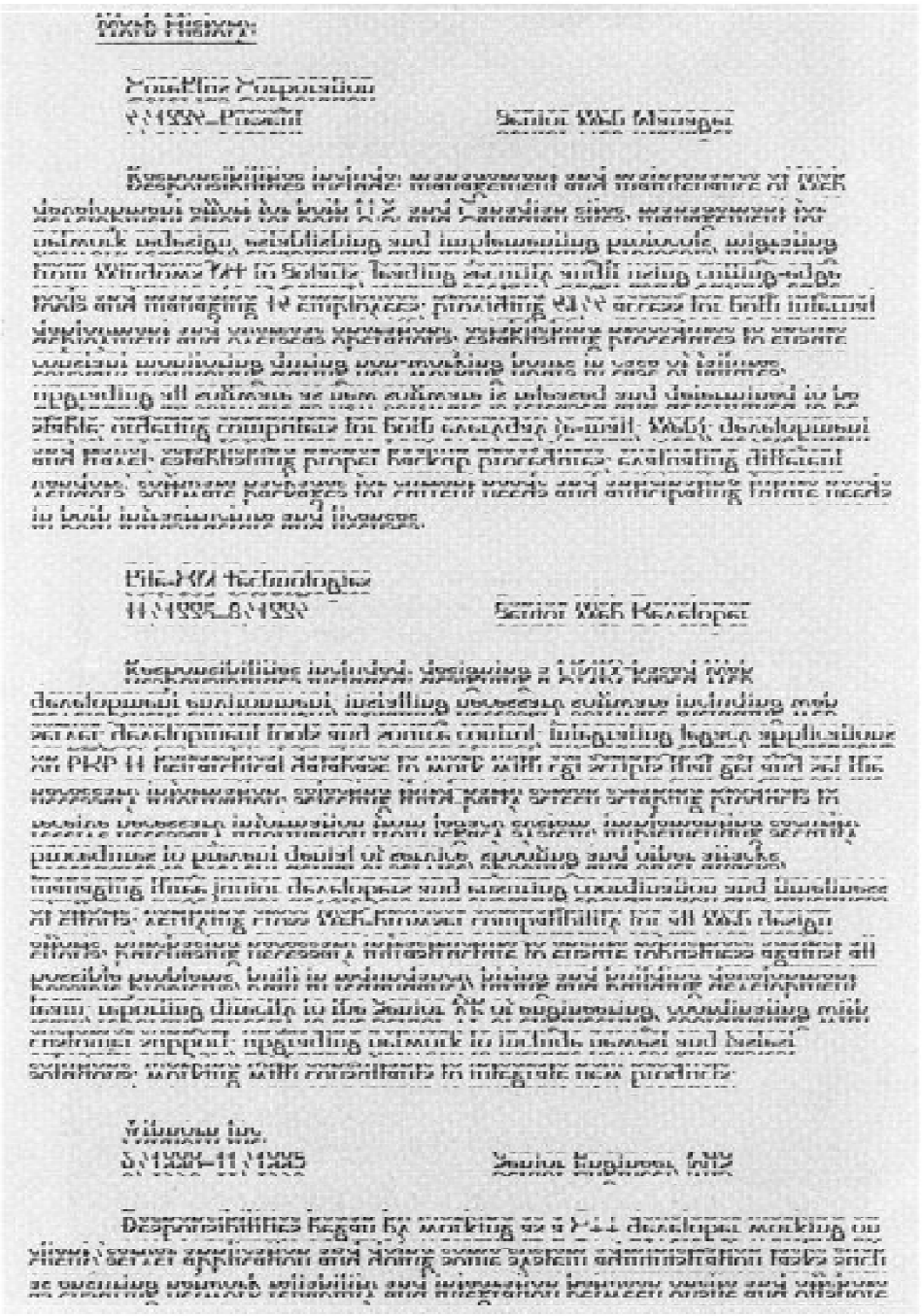
图A-3 润色后的个人简历（初级开发员）



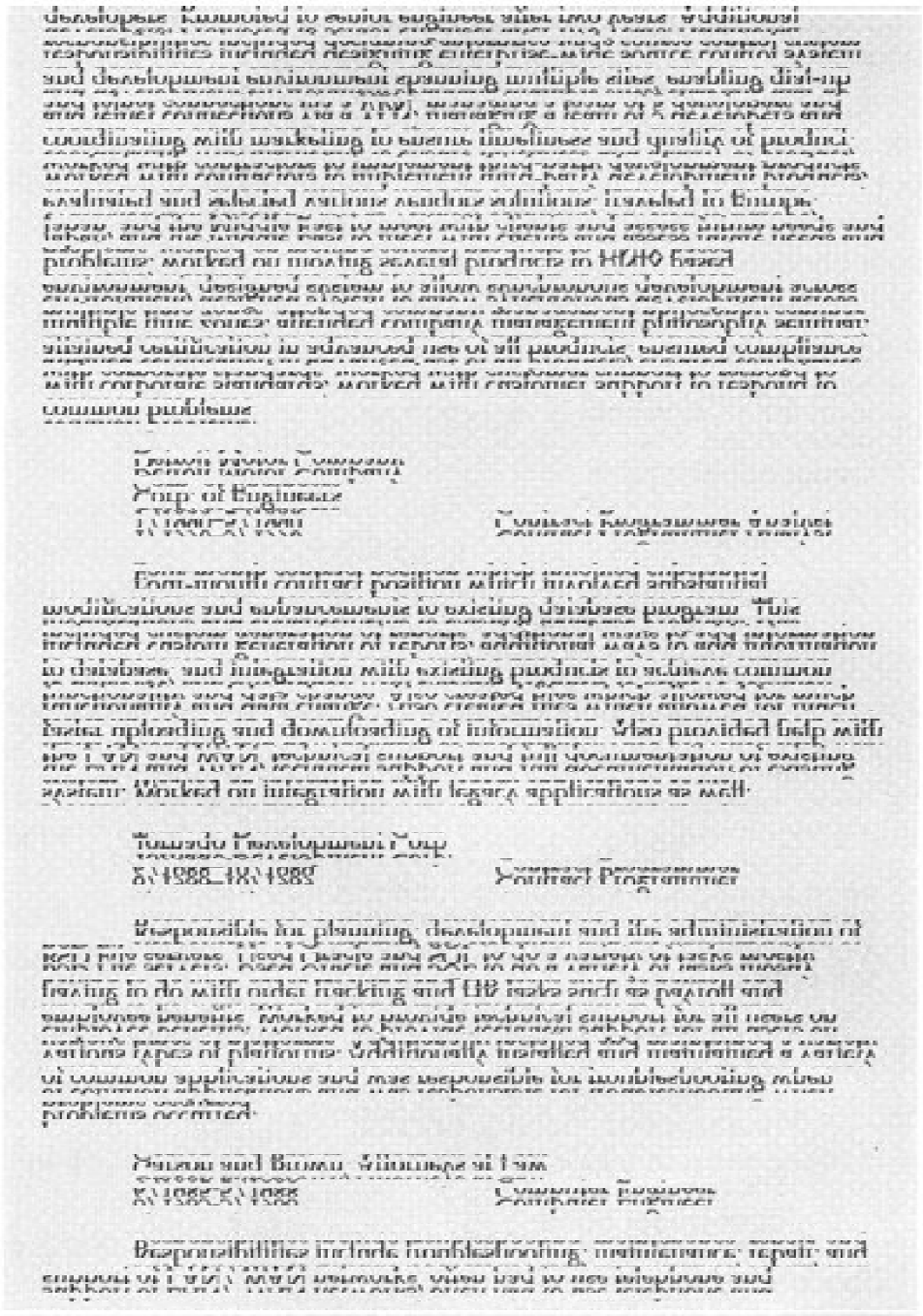
图A-3 （续）



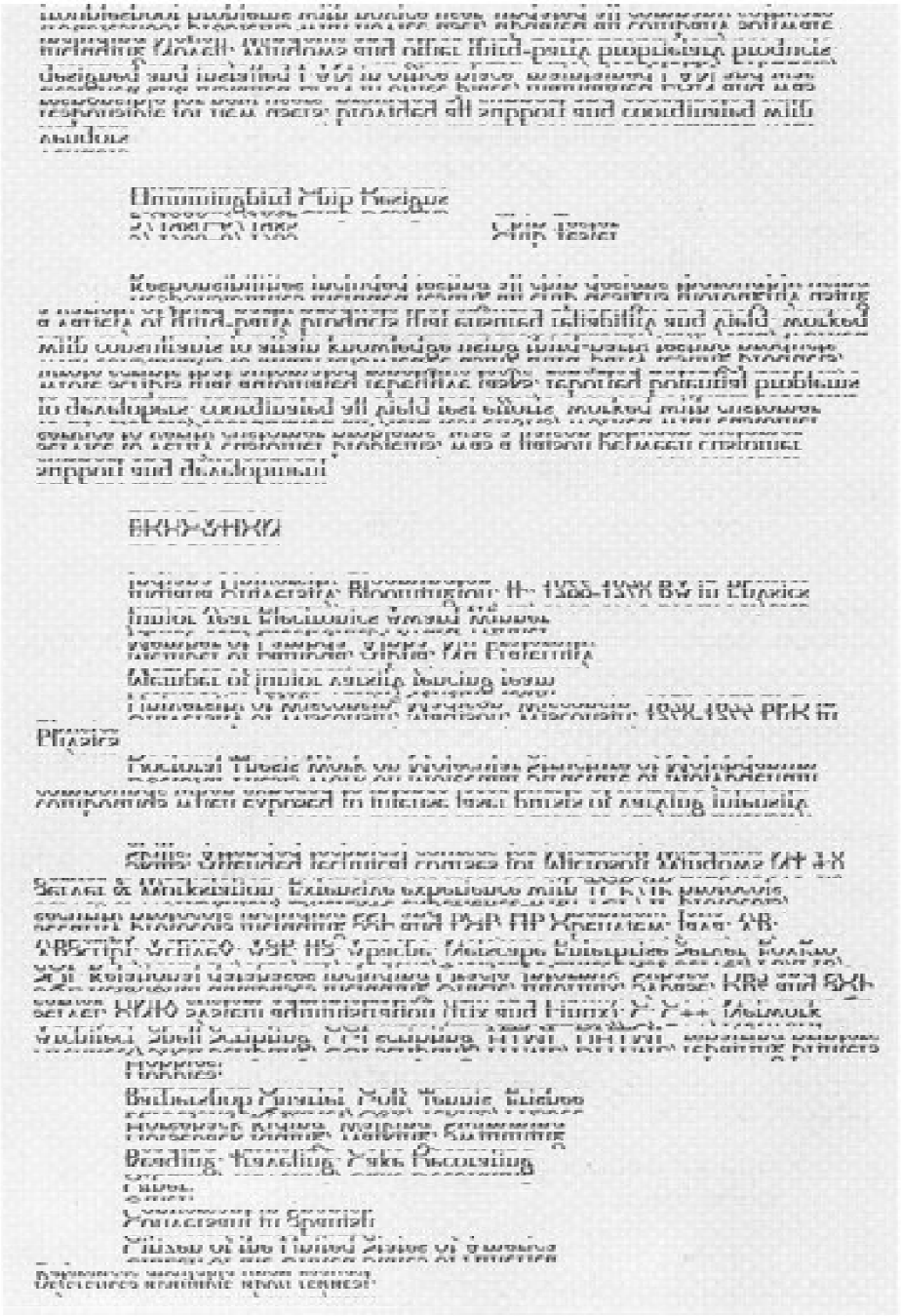
图A-4 润色前的个人简历（项目经理）



图A-4 （续）



图A-4 （续）



图A-4 （续）

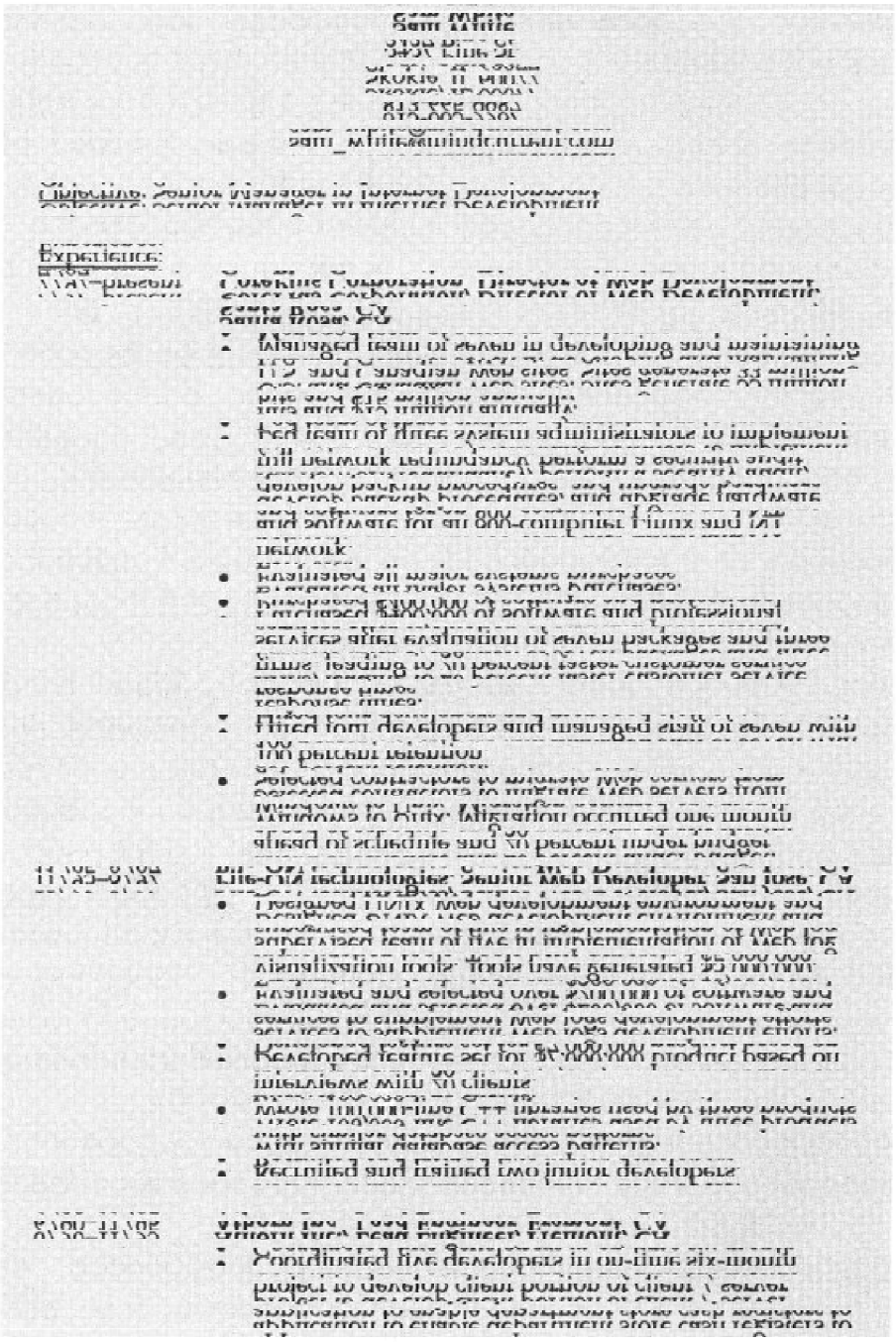
White的个人简历与Lee的第一份简历存在着同样的问题。这是一份自传而不是一个能够把自己推销出去的工具。这一问题从这份简历的开头部分就能看出来：White把自己过去30年的经历都简要地列了出来。把个人简历写成自传是高级职位应聘者们的通病，像White这样拥有辉煌经历的人往往更是如此。很多高级人士相信，只要把自己辉煌的过去写在简历里，面试就会随之而来。与低级职位应聘者一样，这种自信是错误的。除了职位高低有别以外，面试考官脑海里的问题仍然是“这个人能为我们公司干些什么？”在很多情况下，能否做到重点突出对高级人士来说更为重要，因为他们必须在同样短暂的时间内给面试考官留下更深刻的印象。

这份简历里的具体问题也与Lee的第一份简历里的一样。首先，仍然是篇幅太长。White应该把自己个人简历的篇幅压缩到两页以内，最好是一页半左右。White还应该加上项目符号或编号使自己的简历更适于阅读。不过，White最严重的错误却是他的简历与他打算应聘的职务不相吻合。White用了很多篇幅来描述他以前的工作职责，但其中有很多都是初级人士的职责。高级职位通常更重视管理才能，技术方面的能力反而退居其次了。有能力完成初级人士的工作职责并不能保证White具备参加高级职位面试的资格。在申请高级职位的时候，求职者应该重点强调自己的管理才干，而不是强调自己的技术水平或者在初级职位上取得的成绩。

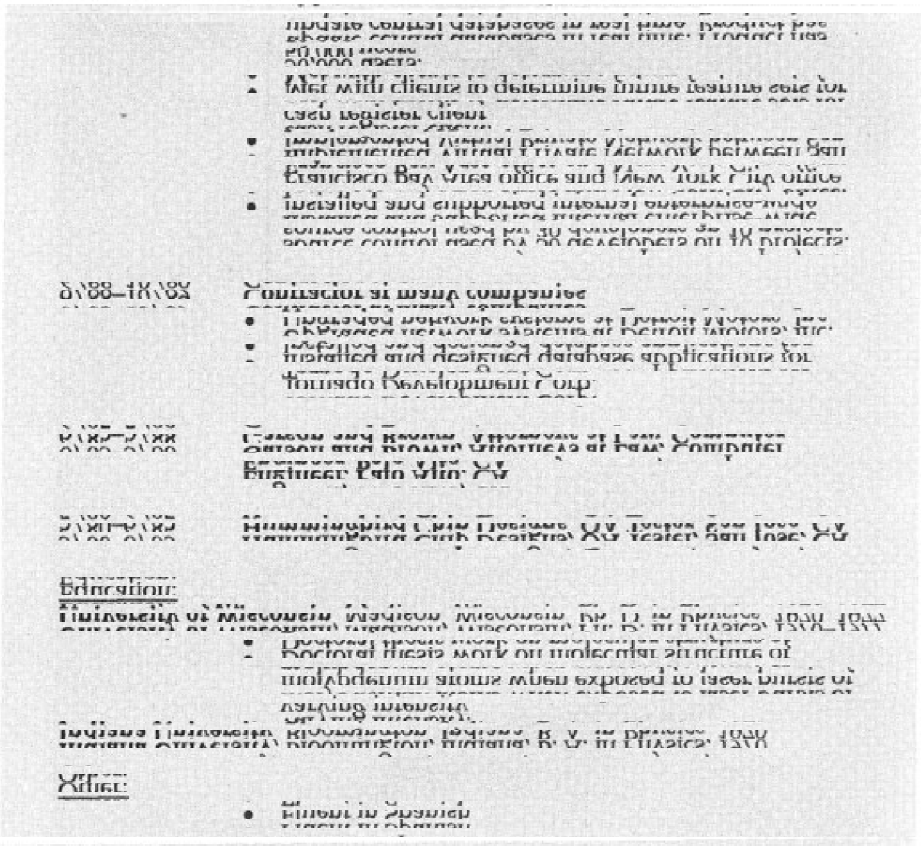
White需要把自己过去的领导才能以及有关成就展示在个人简历里。根据这一思路，我们将对White的管理经历和管理成就进行量化。比如说，White在简历中写道：“management and maintenance of Web development effort for both U.S. and Canadian sites”（负责管理和监督公司在美国和加拿大的站点上的Web开发工作）。这是一项能给人留下深刻印象的业绩，但White既没有提到项目的规模，也没有告诉我们说项目是否成功。White简历中的空白给人留下种种猜测的余地：是不是项目彻底失败而White也不得不引咎辞职？项目是不是太小——只把几份文档放到Web上？White应该尽可能地用数字来证明自己的成就。比如说，他应该这样写：“Managed team of 7 in developing and maintaining U.S. and Canadian Web sites. Sites generate 33 million hits and $15 million annually”（管理一个7人团队，负责管理和监督公司在美国和加拿大的Web站点。这些站点的访问量为3 300万次，年创利润1 500万美元）。

White应聘的职位偏重于项目的管理，计算机技能倒在其次。他应该少用一些口语化的技术词汇，多强调自己在管理方面的经验。他甚至可以考虑少列举几项技术技能——以免招聘方误认为他是想应聘一份不那么高层的职位。

经我们润色和修改后的White个人简历如图A-5所示。在这份新简历里，White在以前工作岗位上的成就更加清晰突出，因而有着更好的推销效果。现在，White已经是一位招聘方无法不邀请来进行面谈的求职者了。

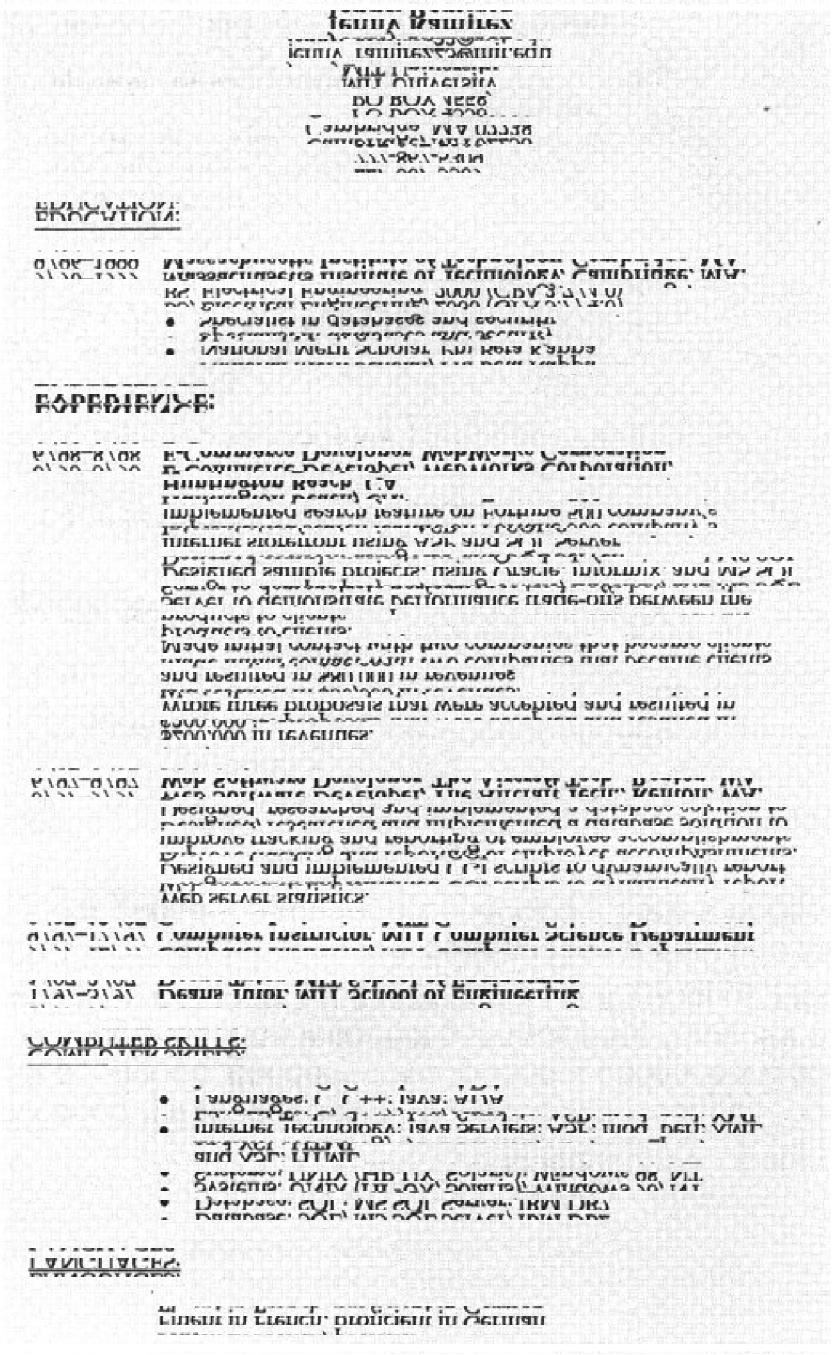


图A-5 润色后的个人简历（项目经理）

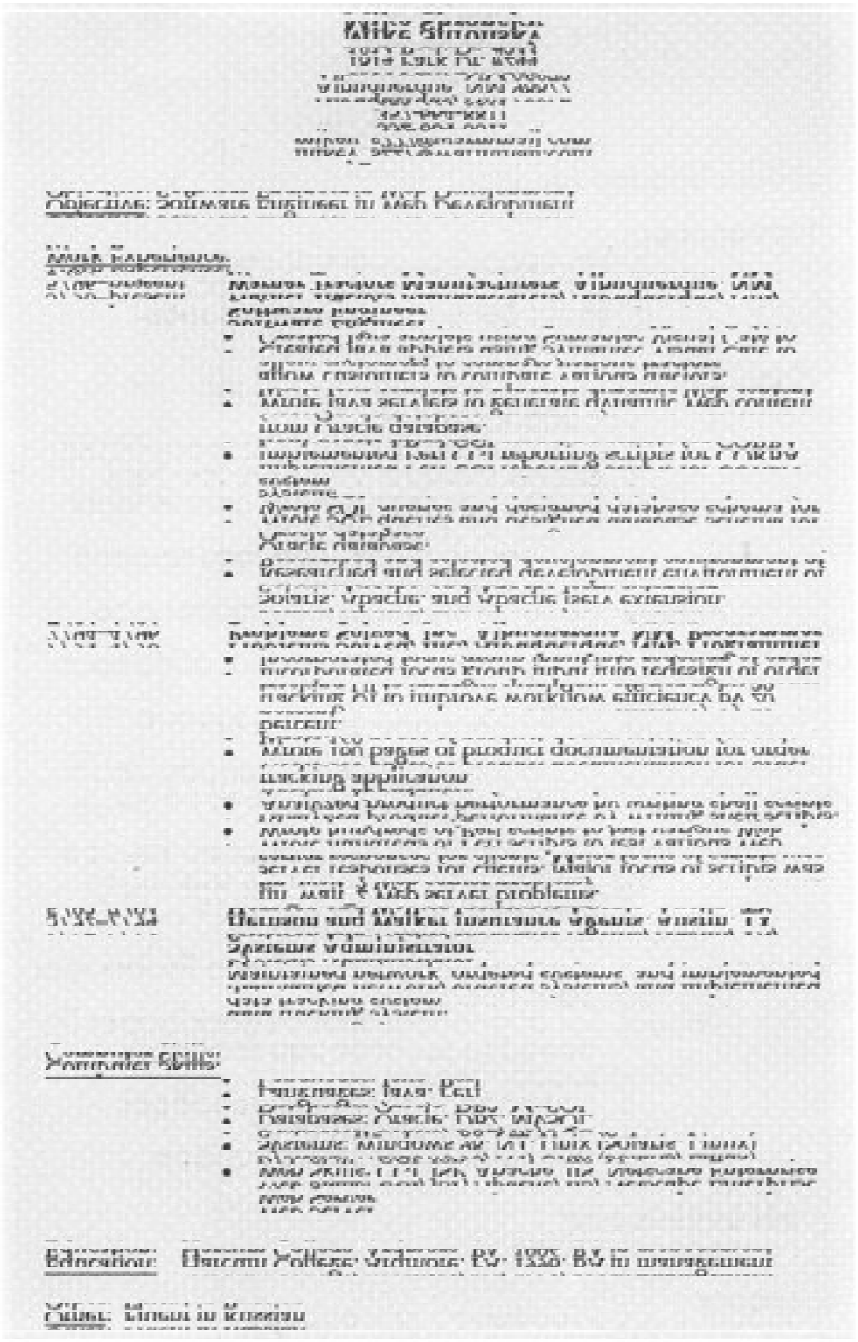


图A-5 （续）

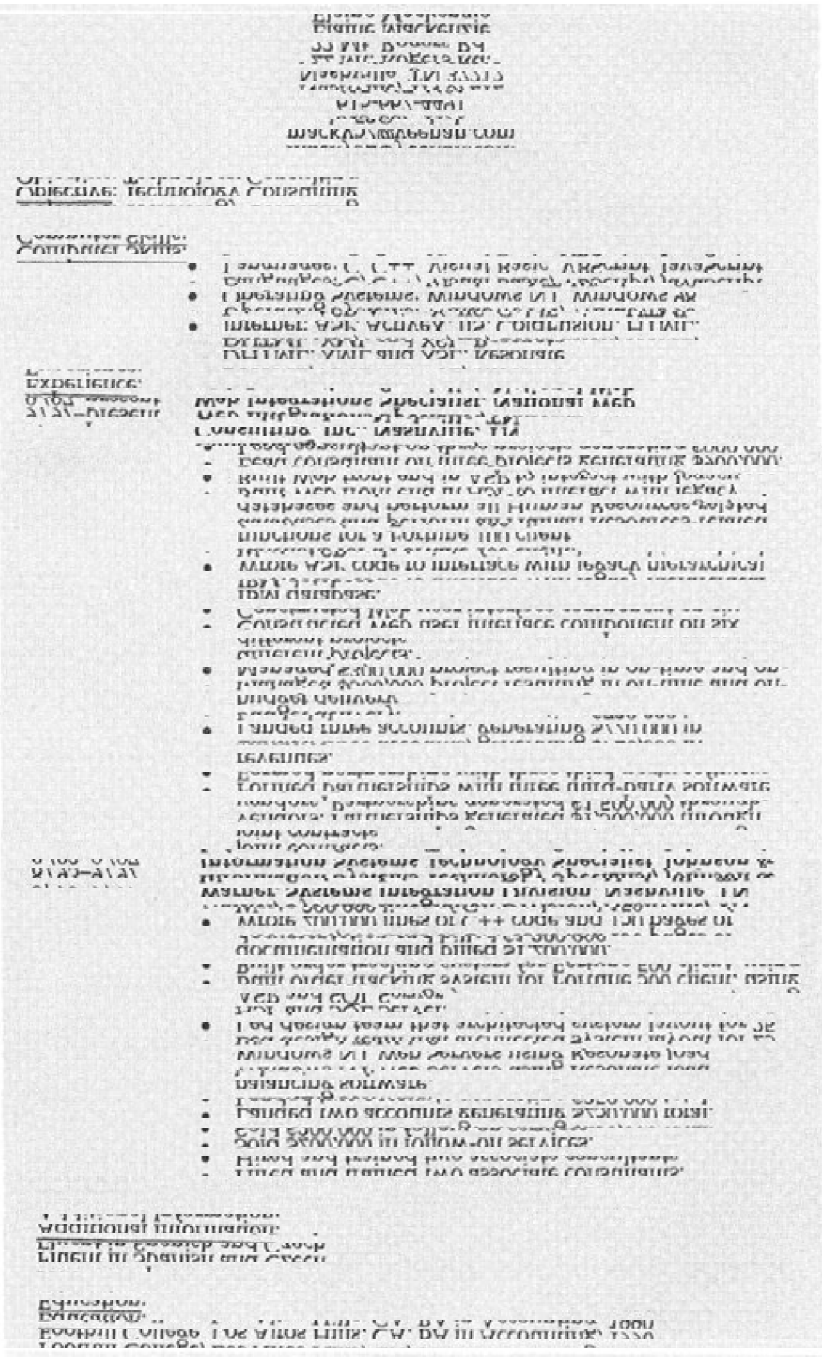
在White的求职道路上，润色后的简历必将成为他自我推销的王牌之一。通过前面两份个人简历上的实战，我们对大家在编写个人简历时可能会出现的许多种失误进行了分析。我们还在下面另外准备了几份写的很不错的个人简历（参见图A-6、图A-7、图A-8），希望大家能够参考它们为自己写出更有效的个人简历来。这三份个人简历的主人公有着不同的工作经验，申请的也是不同的技术类职位。在阅读这几份简历的时候，请大家注意他们都突出了自己的哪些强项，以及这些强项又是如何让他们看起来像是一名有价值的员工的。



图A-6 个人简历示例#1（毕业不久的大学生）



图A-7 个人简历示例#2（Web软件工程师）



图A-8 个人简历示例#3（技术咨询）